I LIBRI di

ioPROGRAMMO

DALLE FONDAMENTA ALLA PROGRAMMAZIONE AVANZATA. IL MANUALE SEMPLICE PER LAVORARE CON IL LINGUAGGIO PIÙ USATO AL MONDO

IMPARARE C++

Roberto Allegra

EDIZIONI MASTER

i libri di

ioProgrammo

IMPARARE C++

Roberto Allegra

A Giabim, che sa perché

EDIZIONI MASTER

INTRODUZIONE

Questo libro ti introdurrà nel mondo del C++: un linguaggio tanto apprezzato e diffuso che non è umanamente possibile tenere il conto dei compilatori che lo supportano, né dei milioni di programmatori che lo utilizzano, né dei libri scritti a riguardo, fra cui quello che hai fra le mani in questo momento. Alcuni di questi testi sono veramente ben strutturati, chiari, e completi - puoi consultare l'appendice per una bibliografia essenziale. "**Programmare in C++, terza edizione**" edito da Addison Wesley [1], in particolare, più familiarmente noto come "**lo Stroustrup**", (dal nome di **Bjarne Stroustrup**: autore del libro, nonché creatore del C++), è decisamente il riferimento assoluto: il Testo Sacro del programmatore C++. È difficile, però, che tu riesca ad apprezzarlo appieno se sei completamente digiuno di C++ o, ancor peggio, di programmazione: si tratta, infatti, di una Bibbia da un migliaio di pagine scritte in modo molto dettagliato, e il C++ è noto per molti meritatissimi pregi, ma non certo per essere un linguaggio semplice da imparare. Intendiamoci subito: è un errore comune sopravvalutarne la difficoltà e la curva di apprendimento. Come ogni prodotto potente e complesso, va affrontato per gradi maturando un po' alla volta la consapevolezza necessaria per sfruttarlo appieno. È quindi ovvio che la finalità di questo testo non è certo quella di renderti un guru capace di districarsi nei mille sentieri aperti dalla conoscenza del C++. Ma alla fine della lettura sarai già produttivo nell'uso del linguaggio, conoscerai le sue potenzialità, e avrai la possibilità di esplorarlo con una mentalità corretta. Io presumerò sempre che tu non abbia mai programmato prima, anche se probabilmente qualche riga di codice l'avrai già scritta. Forse hai addirittura sviluppato per anni in qualche altro linguaggio, e ti sei deciso a "migrare" a C++, per superare vincoli e limiti, avvicinandoti per la prima volta a quel modello OOP (**Object Oriented Programming – vedi capitolo 6**) di cui hai già sentito parlare. In questo libro troverai molte risposte, e qualche domanda. In sintesi:

Il capitolo 1 ti illustrerà la storia di questo linguaggio: da dove viene, perché si usa, e quali sono i programmi necessari per cominciare a scrivere applicazioni.
Il capitolo 2 insegna ad usare i dati primitivi del linguaggio, e combinarli in espressioni.
Il capitolo 3 mostra come gestire il flusso dell'esecuzione con selezioni, cicli e salti.
Il capitolo 4 tratta la creazione di nuovi tipi, come indirizzarli mediante puntatori e riferimenti, ed introduce la memoria dinamica.
Il capitolo 5 spiega come utilizzare il C++ per una programmazione di stampo procedurale, controllando lo spazio dei nomi attraverso i namespaces.
Il capitolo 6 segna la differenza sostanziale fra C e C++, ovvero l'approccio Object Oriented. Imparerai cosa si intende per OOP e come il C++ definisce gli oggetti tramite classi.
Il capitolo 7 specificherà come il C++ applica questi principi: sentirai parlare di ereditarietà multipla, funzioni virtuali e polimorfismo.

Questi argomenti costituiscono la base imprescindibile per programmare in maniera corretta ed efficiente in C++. Nel seguito di questo libro approfondirò, invece, gli aspetti più avanzati: la programmazione generica, le eccezioni, la libreria standard, e alcuni esempi concreti. Con una pratica molto inusuale nei manuali italiani, ma del tutto naturale in altre lingue (ad esempio, in inglese), ho deciso di darti del tu. Spero che questo mi abbia aiutato nel fermo proposito di mantenere sempre un approccio semplice e "hands-on", fuggendo i tecnicismi più insidiosi, e i pomposi e caricaturali pigli cattedratici del tipo "si badi che… al lettore sarà lasciato come esercizio…".

In definitiva, ho voluto intendere questo libro come un'introduzione indolore e alla mano di uno dei linguaggi che più suscitano (a ragione) un sacro timore reverenziale. A questo proposito: se dovessi avere segnalazioni o problemi nel seguire il libro, puoi provare a contattarmi all'indirizzo ImparareIlCpp@yahoo.com.

PER INIZIARE

Nessun linguaggio nasce perfetto, né spunta fuori dal nulla, e il C++ non fa eccezione. Cominciarne lo studio senza avere presenti le coordinate in cui ci si muove porta ad una comprensione raffazzonata, che scaturisce (nel migliore dei casi) in quelle affermazioni dogmatiche e domande raccapriccianti che i neofiti manifestano spesso nei newsgroup o nei forum dedicati al C++.
Ho ritenuto che la forma più leggera per trattare l'argomento fosse proprio una serie di domande di tal fatta (una **FAQ**):

COME SI PRONUNCIA C++?

La pronuncia corretta è "**sì plàs plàs**", all'inglese. Molti, in Italia, lo chiamano comunque "**ci più più**".

QUANDO È STATO INVENTATO?

Il C++ è stato pregettato nei primi anni '80, da Bjarne Stroustrup, con il nome iniziale di **C with classes (C con classi)**. Nel corso di questo ventennio, ovviamente, il C++ ha subito molti cambiamenti! Qui ci dedicheremo alle basi del linguaggio, ma se sei interessato alla sua evoluzione potresti trovare più utile una lettura del libro suggerito in Bibliografia al numero [2], piuttosto che perderti nel marasma degli standard.

COSA VUOL DIRE C++?

Il **C** [3] è il linguaggio creato da Dennis Ritchie intorno agli anni '70, ancor oggi diffusissimo e molto apprezzato per le sue doti di efficienza ed espressività. Proprio per questo Stroustrup ha pensato di fondare su di esso il proprio linguaggio. Come vedrai presto, in C e nei suoi derivati, l'operatore ++ indica l'operazione di incremento. Il nome C++, quindi, vuole suggerire l'idea di un C migliorato. In effetti il C++ è un'**estensione** del C, linguaggio su cui si basa interamente. I compilatori commerciali, in genere, permettono di scrivere sia in C che in C++, e si può sperare che in futuro i due linguaggi diventeranno pienamente compatibili.

ALLORA DEVO CONOSCERE IL C PER IMPARARE IL C++?

Aiuterebbe, ma non è necessario. Anzi: io parto dall'assunto che tu non conosca alcun linguaggio informatico (vedi introduzione).Il C si rivolge ad uno stile di programmazione procedurale, e molti suoi aspetti (vedi domanda successiva) vengono completamente ridefiniti dal C++: non avere una **forma mentis** inquinata dalla programmazione in C, quindi, potrebbe rivelarsi un vantaggio.

IN COSA È MIGLIORATO RISPETTO AL C?

Oltre a controlli molto più stretti sui tipi, e ad alcune differenze di sintassi, il C++ aggiunge diverse parole chiave, ha un meccanismo più sicuro ed intuitivo per la memoria dinamica, e, soprattutto, supporta l'uso delle classi e della programmazione generica. La differenza fondamentale, in effetti, è che per programmare in C++ è richiesta una conoscenza della programmazione a oggetti e dei problemi ad essa correlati.

QUINDI C++ È UN LINGUAGGIO ORIENTATO AD OGGETTI?

C++ non è **puramente** orientato agli oggetti, perché permette una programmazione di stampo procedurale, come quella del C. Il tipico programmatore C++, tuttavia, vede la possibilità di scegliere di non aderire pienamente ad un rigido sistema prefissato come un gran vantaggio (vedi domanda successiva).
Il C++ può essere definito più propriamente come un linguaggio **a paradigma multiplo** (procedurale, a oggetti, e generico).

PERCHÉ DOVREI SCEGLIERE C++ E NON UN ALTRO LINGUAGGIO AD OGGETTI COME JAVA, C#, O EIFFEL?

Il C++ è considerato un linguaggio un po' più complicato rispetto a Java e C#, e l'impianto ad oggetti non regge il confronto con la

purezza offerta dal **design by contract** di Eiffel. Le ragioni principali di tale differenza sono a mio avviso tre:

Il rapporto con la macchina: C++ può diventare, all'occorrenza, un linguaggio di basso livello capace di sporcarsi le mani direttamente con il processore, e tutta la sua architettura è progettata per la vicinanza al calcolatore. Questo si traduce in flessibilità e prestazioni che in linguaggi di più alto livello sono semplicemente irraggiungibili.

C++ non è un linguaggio a oggetti, ma un linguaggio multiparadigma, pertanto presenta un'offerta più vasta e flessibile al programmatore, che deve però possedere una padronanza dei concetti e del linguaggio tali da saper scegliere quella più vantaggiosa nel suo caso specifico.

C++ non impone lo stile da usare nella programmazione con delle regole sintattiche apposite. Questo aspetto viene usato dai detrattori del C++ come motivo di condanna (non avere vincoli rigidi può portare a codice poco leggibile), ma viene invece spesso elogiato da quei programmatori che non vogliono sentirsi **soffocati** da regole coercitive ed inflessibili.

HO SENTITO PARLARE DI MFC, MANAGED EXTENSIONS, CLI... MA QUANTI C++ ESISTONO?

Uno solo, ovviamente. Ma a differenza di linguaggi come VB, Delphi o C#, il C++ **non è un linguaggio proprietario**. Di conseguenza, ogni società è libera di costruire il proprio compilatore (per le architetture più svariate), realizzarlo come vuole, e venderlo – restando attinente, si spera sempre, ad uno standard che permetta la piena compatibilità con quelli degli altri. Diverse case propongono anche delle librerie (come le **Microsoft Foundation Classes o MFC**) per permettere o semplificare la programmazione visuale, o per inserire il linguaggio in un Framework più

ampio, come è il caso delle managed extensions in C++.NET. Noi, qui, **ci limiteremo al C++ standard (ISO)**, che non supporta la programmazione visuale, ma che permette di costruire applicazioni console pienamente funzionanti, e costituisce la base per tutte le estensioni immaginabili.

ESISTONO TANTI COMPILATORI. QUALE DOVREI SCEGLIERE? E QUANTO COSTANO?

Esistono compilatori migliori e peggiori, generici o ottimizzati per il singolo processore, per PC e per sistemi embedded, gratuiti o a pagamento.
In questo libro ci baseremo su GCC (e la sua versione MinGW per Windows), fermo restando che tutto ciò che diremo sarà compatibile per altri compilatori aderenti allo standard ISO.

Setup dell'ambiente di lavoro

Avendo un compilatore e impostando le variabili di ambiente in maniera corretta, è possibile creare programmi col semplice Notepad di windows, o con uno dei diecimila editor che le distribuzioni Linux mettono mediamente a disposizione. Nel corso degli anni, però, sono state sviluppate delle tecniche per rendere la vita più comoda a chi scrive codice, e siccome quella del Programmatore è una categoria pigra per definizione, oggi non molti sarebbero disposti a rinunciare ai vizi offerti da un bell'**IDE** (**Integrated Development Environment -** ambiente di sviluppo integrato). Ogni programmatore ha il suo IDE preferito (o i suoi preferiti), secondo parametri soggettivi che vanno dalla **fede** d'appartenenza (pro o anti windows, etc…), alla presenza/mancanza della caratteristica X (sempre irrinunciabile). Se non hai un IDE installato nel tuo computer, ti consiglio **Code::Blocks** (http://www.codeblocks.org/): un ambiente di sviluppo open source, cross platform, che si aggancia a vari compilatori. Se lavori

sotto il Sistema Operativo Windows, dovrai scaricare la versione **comprensiva di compilatore MinGW** (Minimalist Gnu for Windows: un **porting** del compilatore GCC). Sbrigate le formalità dell'installazione, dovresti trovarti davanti alla schermata rappresentata in (**figura 1.1**).

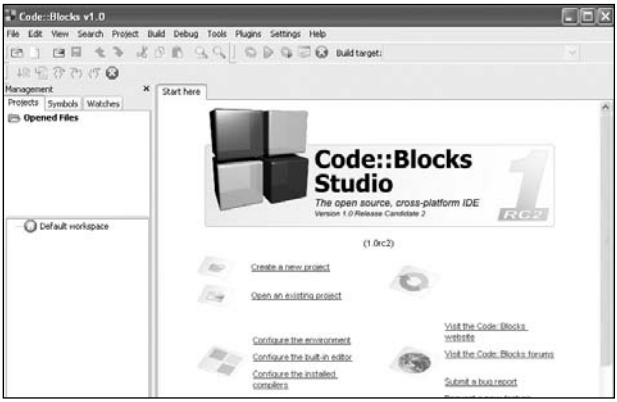

Figura 1.1: L'IDE Code ::Blocks.

Come vedremo ben presto, questo ambiente ci permetterà di scrivere codice, compilarlo, eseguirlo e testarlo (tutto in un uno!) L'IDE provvederà a svolgere per noi tutti i passi di quel processo che deve essere percorso per poter tradurre il codice in un file eseguibile. Per avere una visione corretta di come funziona il C++, però, dovremo partire guardando in dettaglio proprio il funzionamento cruciale di questi passaggi normalmente nascosti.

1.2 IL PROCESSO DI SVILUPPO

Ora che abbiamo impostato tutto, vediamo come funziona il processo che consente ad uno sviluppatore C++ di ottenere la sua applicazione come risultato finale. I passi sono diversi e spesso si compongono di ulteriori suddivisioni interne. Nei paragrafi che seguono analizzeremo il problema secondo il punto di vista schematizzato nella (**figura 1.2**).

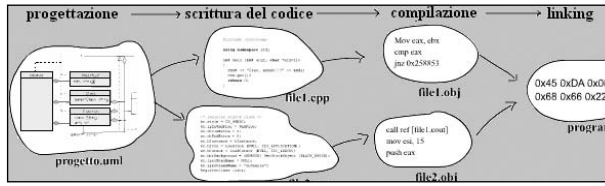

Figura 1.2: Il processo di sviluppo di un'applicazione, dalla progettazione al linking.

1.2.1 CREAZIONE DI UN PROGETTO

Per prima cosa il programmatore pianifica e schematizza il suo lavoro, spendendo tutto il tempo necessario, individuando i bisogni e gli obiettivi dell'applicazione che dovrà costruire. È un passo fondamentale e spesso fin troppo trascurato: vedremo questa fase più in dettaglio nel quinto capitolo. Per ora, per noi il tutto si traduce soltanto nel creare un nuovo **progetto** in Code::Blocks. Un progetto serve ad avere sotto controllo i molti files diversi che sarà necessario scrivere, e che verranno poi riassemblati in un secondo tempo. Per cominciare, allora, apri Code::Blocks e clicca su Nuovo/Progetto. Come puoi vedere in (**figura 1.3**), ti verrà richiesto quale tipo di progetto vuoi realizzare. In questo libro noi analizzeremo una so-

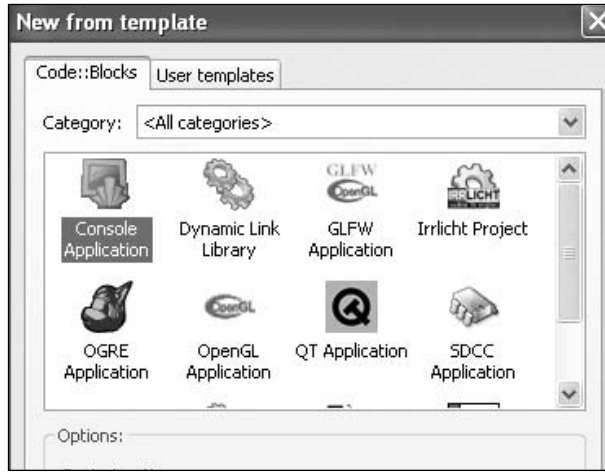

Figura 1.3: Creazione di un nuovo progetto in Code::Blocks

la tipologia, ovverosia le applicazioni di stampo console-based. Nel campo **Nome,** scrivi: "**CiaoMondo**" .Ti verrà presentata una schermata con un piccolo programma già scritto. Qui comincia il secondo (e più impegnativo) passo dello sviluppatore.

1.2.2 SCRITTURA DEL CODICE

Il progetto ha creato automaticamente un file, chiamato main.cpp, in cui è memorizzato il codice predefinito. Non c'è nulla di magico in quanto si trova in questa finestra. Possiamo e dobbiamo modificare il codice sorgente a nostro piacimento! Proviamo a cancellare tutto, e scrivere quanto segue:

```
//Il nostro primo programma!
#include <iostream>

using namespace std;

int main()
{
  cout << "Ciao, mondo!!!" << endl;
  return 0;
}
```


Non preoccuparti se al momento queste linee ti risultano oscure: presto vedremo assieme il loro significato, riga per riga. Per ora, tutto quel che c'è da sapere è che si tratta di codice C++, e che tale codice è corretto. Questo è tutto il nostro programma: stavolta ce la siamo cavata con poco, ma presto i nostri progetti arriveranno a comporsi di più files, e di molte righe di codice.

1.2.3 COMPILAZIONE DEL CODICE

Il programma che abbiamo scritto sarà anche comprensibile per chi conosce il C++, ma non lo è per il freddo elaboratore. Detto in

maniera più tecnica: il C++ è un linguaggio di **alto livello**. Tende, cioè, a trattare i problemi in un'ottica molto astratta, vicina al modello umano di rappresentazione dei problemi (come vedremo: oggetti, classi, strutture), ma decisamente lontana da quella del processore, che opera invece con lunghe sequenze di istruzioni semplici. Per trasformare quello che abbiamo scritto in qualcosa di comprensibile per l'elaboratore, bisogna far ricorso ad un **compilatore**: un traduttore che trasforma i **files sorgente** (quelli scritti in C++, con estensione **cpp)**, in **files oggetto** (con estensione **obj**, oppure **o**). Alla fine della compilazione, quindi, otterremo una serie di files oggetto, tradotti in linguaggio macchina. Code::Blocks usa automaticamente il compilatore che gli viene fornito: nella fattispecie GCC (oppure MinGW) chiamandolo attraverso la riga di comando, proprio come si farebbe a mano.

Fortunatamente l'IDE si occupa anche di inizializzare i riferimenti ai file e alle librerie standard, rendendo la richiesta di compilazione semplice quanto la pressione di un tasto (o un click). Alla fine del processo di compilazione è necessario "ricucire" i vari files .obj, risolvendone le interdipendenze e le chiamate alle librerie esterne. Il programma che svolge questo compito si chiama **linker**, e viene richiamato automaticamente dall'IDE dopo la fase di compilazione. Il risultato è, finalmente, il file eseguibile tanto agognato! Prova a compilare il codice premendo i tasti **CTRL+F9**: verranno creati i files oggetto e il linker comporrà il binario risultante, che puoi eseguire premendo **CTRL+F10**. Spesso vorrai realizzare entrambi i passaggi in una volta sola: basterà premere il tasto **F9**.

1.3 CIAO, MONDO!

Il risultato dell'esecuzione è visibile in (**figura 1.4**). L'applicazione gira sulla console testuale, che in Windows viene aperta per l'occasione nella finestra di emulazione MS-DOS. A questo punto, come promesso, riprendiamo in mano il codice, spiegandolo riga per riga. Non è mia intenzione farti capire adesso il significato più intimo di ogni istruzione, ma è indis-

Figura 1.4: Output dell'esecuzione di

pensabile che tu ti faccia un'idea su come è strutturato un programma tipico. Nel corso dell'analisi vedremo anche alcuni concetti chiave, come i commenti, il preprocessore e l'uso degli stream predefiniti.

1.3.1 IL NOSTRO PRIMO PROGRAMMA

La prima riga ci mostra una caratteristica tipica degli editor di codice: il syntax highlighting, ovverosia la diversa colorazione del testo a seconda del relativo valore sintattico. Code::Blocks colora questa prima istruzione di grigio, colore che riserva ai commenti. Un commento è una riga di codice che non sarà presa in considerazione dal compilatore, ma che è utile come promemoria per i programmatori che leggeranno quanto scritto: saper commentare efficacemente il proprio lavoro è uno dei punti che distinguono i buoni programmatori dalla massa degli "scrittori di codice". Il C++ permette due tipi di commenti. Quello usato per questa linea (una doppia sbarra) è il commento a riga singola, che ordina al compilatore di saltare tutto ciò che si trova alla destra delle sbarre, fino al primo ritorno a capo. Un secondo tipo di commento (mutuato dal C) è quello "a blocchi", che inizia con la sequenza "/*" e termina con "*/", e che può estendersi su più righe.

1.3.2 #INCLUDE <IOSTREAM>

Questa seconda riga è segnalata in verde. Tale colore indica le direttive rivolte al preprocessore: un programma richiamato durante il primissimo stadio della compilazione, e che ha lo scopo di sostituire i riferimenti a macro e file header esterni, con i valori relativi (nonché di saltare i commenti). L'istruzione #include <nome>, in particolare, serve ad includere (cioè a copiare letteralmente) un file qualsiasi. I files inclusi si chiamano headers (intestazioni), e hanno lo scopo di dichiarare funzioni e variabili esterne. Sono, in altre parole, la "porta di accesso" che permette al nostro programma di usare le funzionalità presenti in librerie esterne. Il file in questione (iostream, che puoi trovare nella sottocartella include/c++/3.x.x), fa parte della libreria standard del C++, e contiene le definizioni necessarie a gestire i flussi di ingresso e di uscita (vedremo meglio la definizione di stream più avanti): includerlo ci permetterà di usare due variabili esterne fondamentali:

std::cout: serve a scrivere sul canale di uscita primario (solitamente, la finestra della console)
std::cin: serve a ricevere dal canale d'ingresso primario (solitamente, la tastiera).

1.3.3 USING NAMESPACE STD;

Anche se vedremo bene il concetto più avanti, per ora puoi vedere i namespace come dei contenitori che racchiudono nomi di variabili e funzioni, in maniera da distribuire il codice opportunamente e risolvere ambiguità (un po' come le directory usate dai sistemi operativi). E' possibile accedere ad una variabile o ad una funzione dichiarata in un namespace attraverso il simbolo "::". Cin e cout, ad esempio, fanno parte del namespace std, in cui sono racchiusi nomi e funzioni della libreria standard del C++, e pertanto andrebbero richiamati, ogni volta, scrivendo "std::cin" e "std::cout". Ma, se torni a guardare il codice, vedrai che non l'abbiamo fatto!

Il merito è proprio dell'istruzione "using namespace std;", che indica al compilatore di **dare per scontato che vogliamo riferirci a variabili appartenenti al namespace std**: ciò rende tutto più comodo.

1.3.4 INT MAIN() { … }

Questa riga indica l'inizio della funzione principale (main), detta anche punto d'ingresso dell'applicazione. Scriverla è obbligatorio, e sarebbe insensato fare altrimenti, dal momento la funzione main specifica quel codice che dovrà essere eseguito all'avvio del programma.

Questa riga, inoltre, specifica un valore di ritorno di tipo intero (int). La funzione principale, cioè, dovrà restituire, alla fine dell'esecuzione, a un valore intero, che viene in genere usato dal sistema, o dal programma chiamante, per sapere se l'esecuzione è andata a buon fine, o meno. Le parentesi graffe che racchiudono il blocco d'istruzioni che seguono sono anch'esse obbligatorie, e servono ad indicare l'inizio e la fine della funzione.

1.3.5 COUT << "CIAO, MONDO!!!" << ENDL;

Questa riga è la nostra prima vera istruzione. Come abbiamo già visto nel paragrafo 1.4.1, la variabile cout appartiene al namespace std, e ci viene fornita dall'inclusione dell'header <iostream>. Semplificando molto, puoi vederla come "lo schermo". Ogni volta che vorrai inserire dei dati nello schermo, potrai seguire la sintassi:

```
cout << dato1 << dato2 << daton;
```

L'operatore '<<' (inserimento) serve ad inserire uno stream in un altro. Dal momento che anche i dati vengono convertiti in stream a loro volta, l'operazione è facilmente cumulabile e permette la sintassi scritta sopra. Il dato in questione, in questo caso, è una stringa

("Ciao, mondo!!!"): una sequenza di caratteri, che va inizializzata sempre fra virgolette.

Endl è un manipolatore esterno appartenente a std, e serve ad inserire una sequenza di fine riga (andare a capo). Per finire, non va dimenticato il punto e virgola, che serve ad indicare che l'istruzione è terminata.

1.3.6 RETURN 0;

Infine, l'ultima riga: dal momento che abbiamo dichiarato che la funzione main deve restituire un valore intero, dobbiamo mantenere la promessa. Come vedremo, la parola chiave return serve proprio a questo scopo: interrompe l'esecuzione della funzione, e restituisce il valore indicato come parametro. In questo caso, 0, che indica un'esecuzione andata a buon fine.

1.4 CONCLUSIONI

In questo primo capitolo abbiamo accennato a moltissime cose, che vedremo molto più compiutamente in futuro. Se quindi ti stai preoccupando perché credi di non aver capito cos'è perfettamente uno stream, o un tipo di dato, voglio tranquillizzarti: sarebbe strano il contrario. Ciò che è necessario per continuare, è che tu abbia:

- Installato correttamente l'IDE Code::Blocks, e fatto girare il tuo primo programma.
- Imparato chiaramente il processo: scrittura -> compilazione -> esecuzione

- Imparato come creare un nuovo progetto, come scrivere codice, e la struttura tipica di un programma C++.

- Imparato come usare la variabile **cout** per scrivere dati sullo schermo

- Imparato come usare la variabile **cin** per bloccare il flusso dell'esecuzione.

Dal prossimo capitolo imparerai a scrivere applicazioni più complesse.

1.5 PROGETTI ED ESERCIZI

Alla fine di ogni capitolo, ti proporrò esercizi e progetti da realizzare da solo. In questo caso, date le scarse conoscenze finora accumulate, cominciamo con un compito molto semplice:

Scrivere un programma che stampi su tre linee distinte, le parole "casa dolce casa"

Estensione: Di quante istruzioni di tipo "cout << dato;" ti sei servito? Sapresti scrivere tutte e tre le righe con un'istruzione sola?

DATI ED ESPRESSIONI

Per ora siamo in grado di usare la console alla stregua di una macchina per scrivere: alla fine di questo capitolo sapremo costruire programmi ben più complessi. Impareremo i tipi di dato fondamentali, la possibilità di ricevere input dall'utente, l'elaborazione di espressioni. Probabilmente tutto ciò rende questo capitolo il più importante e impegnativo. Se sei a digiuno di programmazione, prenditi molto tempo per studiarlo bene e svolgere gli esercizi proposti.

2.1 TIPI DI DATO FONDAMENTALI

La programmazione OOP, le cui basi teoriche vedremo nel capitolo 5, si basa sul concetto espresso nella citazione: ogni ente è rappresentabile attraverso degli oggetti, che possono essere visti come una composizione di dati più semplici. Alla base di ogni oggetto, quindi, direttamente o meno, ci sono sempre dei tipi atomici, chiamati **primitivi**, che il C++ fornisce per la descrizione di dati fondamentali (caratteri, numeri interi, numeri decimali e valori logici): avviarsi sulla strada del C++ senza conoscerli sarebbe un grave errore! Qui di seguito definisco i vari tipi di dato, e nel paragrafo successivo vedremo come questi possono essere usati per i nostri scopi pratici.

2.1.1

Variabili: le variabili booleane (dichiarate con la parola chiave **bool,** pronuncia **bul)**, si usano per indicare un dato che può avere solo due stati: **true** (vero), oppure **false** (falso). Per rappresentare i dati bool sarebbe sufficiente un bit, tuttavia (a meno di ottimizzazioni particolari) i calcolatori sono costretti a riservar loro comunque l'unità minima di allocazione, che è solitamente 1 byte.
Costanti: Nella conversione a intero, **true** equivale a 1 e **false** equivale a 0: in effetti, in C++ - così come in C - la verità logica si esprime con ogni valore diverso da 0.

2.1.3 INTERO

Valori: Gli interi (parola chiave **int**) rappresentano i numeri naturali in un raggio che varia a seconda del numero di bytes riservati a questo tipo di dato (minimo 2, in genere 4). È possibile allungare o accorciare tale raggio con dei modificatori:

short (o short int), indica di utilizzare, se possibile, un raggio più corto.

long (o long int), indica di utilizzare, se possibile, un raggio più lungo

Normalmente gli interi sono **signed,** e, anche se solitamente non è una pratica molto ben vista, è possibile dichiarare un **unsigned int, short o long**, per utilizzare anche il bit riservato al segno.

Costanti: Le costanti di tipo intero possono essere scritte semplicemente indicando il numero corrispondente. È anche possibile scrivere il numero in esadecimale, anteponendo i due caratteri '0x', e in ottale, anteponendo il numero 0. Il numero 64, ad esempio, può essere scritto in ottale come 0100, e in esadecimale come 0x40.

2.1.4 DECIMALE

Valori: I numeri decimali vengono rappresentati in C++ attraverso il sistema della virgola mobile. Sono possibili tre tipi di dichiarazione:

float: indica un numero a precisione singola

double: indica un numero a precisione doppia

long double: indica un numero a precisione estesa.

Come al solito, il numero di bytes, il raggio dei valori, nonché l'esatto significato dei termini "precisione doppia" e "precisione estesa", dipendono completamente dall'interpretazione che il compilatore ne dà.

Costanti: Una costante può essere dichiarata come decimale in diversi modi. Il più semplice è quello di porre il punto decimale, anche quando si tratta di cifre di per sé intere: (3.1415, 16.0, etc…). Un uso sconsiderato delle costanti può portare ad effetti imprevisti: il risul-

tato di 3.0/2.0 è 1.5, mentre 3/2, in quanto divisione fra interi, risulta 1 (col resto di 2)!

2.2 DICHIARAZIONE, INIZIALIZZAZIONE E ASSEGNAMENTO

Ora che abbiamo visto i principali tipi di dato e le relative parole chiave, dobbiamo riuscire ad usarle attivamente nel codice: per questo dobbiamo introdurre il concetto di **variabile:** una zona di memoria in cui è presente un dato di un tipo ben definito, che è possibile modificare durante il corso dell'esecuzione. Per poter usare una variabile è necessario:

- **Darle un nome**, ovverosia un identificativo. Per essere legale, un ID dev'essere una sequenza alfanumerica iniziante per lettera - **a, a1, c2d3,** ad esempio.

- **Stabilire il tipo cui appartiene**: questa scelta va fatta subito, e non è possibile cambiarla durante il corso del programma.

Per comunicare queste scelte al calcolatore è necessario **dichiarare** la variabile in un punto qualsiasi del programma.
La dichiarazione viene fatta secondo la sintassi:

```
tipo variabile;
```

Proviamo a vedere questo programma:

```
//Dichiarazioni
#include <iostream>

using namespace std;
```

```
int main() {
   long x;      //tipo = long, variabile = x
   int  y; //tipo = int,  variabile = y
   char z;      //tipo = long, variabile = z

   cout << "Valore di x: " << x << endl;
   cout << "Valore di y: " << y << endl;
   cout << "Valore di z: " << z << endl;
   return 0;
}
```

Con questo codice abbiamo dichiarato nella funzione principale tre variabili: x, y e z, di tre tipi differenti. Quale sarà l'output generato dal programma?
Non c'è alcuna possibilità di saperlo! In uno scenario plausibile, lo stravagante risultato dell'esecuzione di questo programma potrebbe essere:

```
Valore di x: 3
Valore di y: 62
Valore di z: [
```

Puoi capire il perché di questi strani valori ripensando alla definizione di variabile: un semplice nome associato a un dato contenuto in uno specifico indirizzo di memoria. Questi valori sono esattamente ciò che si trova in queste celle, **sporcate** dell'esecuzione di programmi precedenti. A differenza di altri linguaggi di programmazione, infatti, il C++ non inizializza da solo la memoria ponendola a zero, o su un valore neutro: e questo, per inciso, è un tipico esempio di quell'attenzione che questo linguaggio dà alle performance - in fin dei conti, è tutto tempo risparmiato! Molto spesso, quindi, oltre a dichiarare una variabile ti ritroverai a volerla **inizializzare**, ovverosia a volerle dare un

valore iniziale. Il C++ offre questa possibilità attraverso un ampliamento della sintassi della dichiarazione:

```
tipo variabile = espressione;
```

A titolo di esempio, puoi guardare il codice seguente:

```
//Inizializzazioni a riga multipla
#include <iostream>

using namespace std;

int main() {
char lettera1 = 'C';
char lettera2 = 'I';
char lettera3 = 'A';
char lettera4 = 'O';
cout      << lettera1
<< lettera2
<< lettera3
<< lettera4;
return 0;
}
```

In questo caso, l'output sarà

```
CIAO
```

La sintassi, infine, prevede un'ulteriore ampliamento nel caso in cui si vogliano dichiarare più variabili dello stesso tipo su una sola riga:

```
tipo variabile1 [= espressione1], variabile2 [= espressione2] …;
```

Laddove con le parentesi quadre intendo il fatto che l'inizializzazione dei valori è facoltativa. Le dichiarazioni dell'esempio precedente possono essere così riportate in una forma più compatta:

```
//inizializzazione a riga singola
char lettera1 = 'C', lettera2 = 'I', lettera3 = 'A', lettera4 = 'O';
```

oppure, rendendo il tutto più leggibile:

```
char      lettera1 = 'C',
lettera2 = 'I',
lettera3 = 'A',
lettera4 = 'O';
```

le due scritture sono assolutamente identiche, dal momento che il ritorno a capo in C++ viene ignorato (ho usato lo stesso principio anche per separare i vari valori inseriti in cout nell'ultimo codice). Spesso non è dato sapere al momento della dichiarazione il valore che assumerà la variabile, e questo può comunque cambiare durante il corso del programma. In qualsiasi momento, pertanto, è possibile effettuare un **assegnamento** su una variabile già dichiarata, secondo la semplice sintassi:

```
variabile = espressione
```

come riferimento pratico, puoi analizzare l'esempio seguente:

```
//Assegnamenti
#include <iostream>
using namespace std;
int main() {
int v = 1;
```

```
    cout << "all'inizio, v vale: " << v << endl;

    v = 2;
    cout << "dopo il primo assegnamento, vale: " << v << endl;

    v = 100;
    cout << "e dopo il secondo: " << v << endl;

    return 0;
}
```

L'ovvio risultato dell'esecuzione di questo codice sarà

```
All'inizio, v vale: 1
dopo il primo assegnamento, vale: 2
e dopo il secondo: 100
```

2.3 L'OPERAZIONE DI ESTRAZIONE TRAMITE CIN

Nel paragrafo precedente abbiamo visto come le inizializzazioni e gli assegnamenti servano ad associare un valore alle variabili. Tale valore può essere una costante o un'espressione, ma è comunque frutto di quanto stabilito nel codice. A volte, però, vorremo chiedere all'utente il valore di una determinata variabile: in fin dei conti, sono ben pochi i programmi che non prevedono un minimo di interazione con chi sta dall'altra parte dello schermo! Per richiedere un valore possiamo usare lo stream std::cin, che rappresenta il flusso dati in ingresso, solitamente associato alla tastiera. Finora abbiamo usato il flusso std::cout, servendoci dell'operatore di **inserimento** (<<), che permette di agganciare uno stream alla fine di un altro. L'operazione inversa si chiama **estrazione (>>)**, e permette di estrarre dei dati da uno stream. Con un sistema che

può sembrare un po' magico, ma che in realtà è frutto di un sapiente **sovraccaricamento degli operatori**, l'estrazione cerca di convertire il dato estratto, coerentemente con il tipo di variabile in cui tale dato verrà inserito.
La sintassi dell'estrazione è:

```
stream >> variabile;
```

Dal momento che in questo testo analizzeremo prevalentemente cin come stream d'ingresso, troveremo sempre:

```
cin >> variabile;
```

Qui di seguito presento un esempio sull'utilizzo tipico di cin per l'estrazione di un valore:

```
//Uso di cin per l'estrazione
#include <iostream>
using namespace std;
int main() {

cout << "Scrivi un numero";
int a;
cin >> a;

cout << "Il numero che hai scritto è: " << a;

    return 0;
}
```

2.4 ESPRESSIONI ARITMETICHE

Sappiamo descrivere un dato, dichiararlo, assegnarlo ad una variabile e memorizzare un valore richiesto dall'utente. Il prob-

lema è che non abbiamo ancora stabilito alcuna relazione fra i dati in nostro possesso. Un'**espressione** è una combinazione di più valori effettuata attraverso uno o più **operatori.**

Un insieme di comprensione immediata è quello degli **operatori aritmetici**, presentati in tabella 2.1, che include gli operatori comunemente utilizzati sulle calcolatrici tascabili, più quelli di shift, che meritano un approfondimento. Abbiamo già visto gli operatori "<<" e ">>" con il significato rispettivamente di **inserimento** ed **estrazione**: questo è vero soltanto quando tali operatori si applicano agli stream (come cin e cout), perché il loro significato originale viene ridefinito (in gergo: **sovraccaricato**).

Per i dati primitivi, invece, questi operatori prendono il nome di **bit shifting sinistro (<<), e destro (>>)** hanno la funzione di spostare i bit che compongono il dato, nella rispettiva direzione, eliminando quelli che "finiscono fuori" e rimpiazzando quelli mancanti con 0.

L'espressione **48 >> 3**, ad esempio, indica di spostare i bit del numero 48 (ovvero 00110000), di tre posizioni a destra, dando così come risultato 6 (ovvero 00000110). Matematicamente, si può esprimere **n << m** come $\mathbf{n * 2^m}$, e **n >> m** come $\mathbf{n / 2^m}$, a meno di possibili **overflow.**

Simbolo	Operazione
>>	Bitshift destro
<<	Bitshift sinistro
*	Moltiplicazione
/	Divisione
%	Modulo (resto della divisione)
+	Addizione
-	Sottrazione

Tabella 2.1: Operatori aritmetici

Gli operatori, inoltre, lavorano secondo un preciso ordine di precedenza, che ricalca l'ordine in cui figurano nella Tabella 2.1. Per alterarlo è necessario usare le parentesi tonde, come si può notare nel seguente esempio.

```
int x = 3 + 3 * 5 * 8 – 3          // x = 120
int y = ((3 + 3) * 5) * (8 – 3)    // y = 150
```

Ovviamente, ha poco senso lavorare con delle espressioni interamente costanti, dal momento che ne conosciamo a priori il risultato: in questi casi basterebbe sostituire il codice riportato precedentemente con i relativi commenti!

2.4 PRATICA: UN PROGRAMMA CALCOLATORE

In questo paragrafo potrai consolidare quanto acquisito finora, scrivendo un programma che funga da "calcolatrice tascabile", intesa nella sua versione più semplice: la nostra applicazione chiederà due numeri in ingresso all'utente, e stamperà a video la rispettiva somma, differenza, etc…Puoi provare a realizzarlo da solo, prima di guardare il codice che è riportato qui di seguito.

```
//Calcolatrice minima
#include <iostream>
using namespace std;
int main() {
int a, b;
//richiesta dei due numeri
cout << "Scrivi il valore di a: ";
cin >> a;
cout << "Scrivi il valore di b: ";
cin >> b;
```

```
//stampa dei risultati

cout      << a << " +  " << b << " = " << a + b    << endl
          << a << " -  " << b << " = " << a - b    << endl
<< a << " *  " << b << " = " << a * b    << endl
<< a << " /  " << b << " = " << a / b    << endl
<< a << " %  " << b << " = " << a % b    << endl
<< a << " << " << b << " = " << (a << b) << endl
<< a << " >> " << b << " = " << (a >> b) << endl;
return 0;
}
```

L'output dell'esecuzione, dati in ingresso a = 200 e b = 5, è:

```
Scrivi il valore di a: 200
Scrivi il valore di b: 5
200 +  5 = 205
200 -  5 = 195
200 *  5 = 1000
200 /  5 = 40
200 %  5 = 0
200 << 5 = 6400
200 >> 5 = 6
```

La comprensione del programma dovrebbe essere immediata: l'unico punto non banale è l'uso delle parentesi tonde nelle espressioni di shift, che in questo caso è necessario per evitare ambiguità con gli operatori sovraccaricati. Questo è evidente nel caso dello shift sinistro.

```
cout << 10 << 1;      // inserimento;
cout << (10 << 1);    // bitshift;
```

Nel primo caso sarà stampato a video 101, che è la concatenazione

di 10 e 1, e nel secondo 20, che è il risultato dell'operazione di shifting. Ciò avviene perché quando i dati vengono convertiti in stream, come nel caso di una chiamata a cout, il C++ ridefinisce automaticamente il significato originario degli operatori di shift con quello sovraccarico di "estrazione" o "inserimento".

2.5 OPERATORI RELAZIONALI E LOGICI

Difficilmente immagineremmo di poter scrivere un programma qualsiasi senza usare le espressioni aritmetiche viste nel paragrafo 2.4. Altrettanto fondamentali sono quelle che valutano il confronto fra due espressioni, realizzato mediante uno degli operatori elencati in (**Tabella 2.2**)

Simbolo	Operazione
==	Uguale a
!=	Diverso da
<	Minore di
>	Maggiore di
<=	Minore o uguale a
>=	Maggiore o uguale a

Tabella 2.2: Operatori relazionali

Nota: Fa' attenzione all'operatore di uguaglianza (==): è uno dei punti critici in cui cadono tutti i neofiti, i quali tendono a confonderlo spesso e volentieri con l'operatore di assegnamento (=), ottenendo così programmi malfunzionanti.

Queste espressioni restituiscono un valore bool (vedi paragrafo 2.1.1) che indica se l'affermazione dichiarata è vera o falsa. Il risultato del codice:

```
cout << "3 e' minore di 2? "  << (3 < 2) << endl;
```

```
cout << "3 e' maggiore di 2?" << (3 > 2) << endl;
```

è

```
3 è minore di due? 0
3 è maggiore di due? 1
```

Il che è quanto ci si aspetta, dal momento che – conviene ripeterlo – 0 indica falso e 1, così come ogni altro valore, indica vero. Spesso, tuttavia, si ha la necessità di dover operare su più espressioni booleane, messe in relazione fra loro: questo compito viene svolto per mezzo degli operatori logici.

Simbolo	Operazione	a	b	r
!	Not	0 1		
&&	And	0 0 1 1	0 1 0 1	0 1 0 1
||	Or	0 0 1 1	0 1 0 1	0 1 0 1

Tabella 2.3: Operatori logici

In (**tabella 2.3**) sono elencati i tre operatori logici presenti in C++, con le relative tavole di verità, che mostrano il risultato al variare dei valori a e b.

- **L'operatore not (!)**, unario, si usa per invertire un valore logico: il risultato è pertanto vero quando il valore in ingresso è falso, e viceversa.

- **L'operatore and (&&)**, restituisce vero solo quando entrambi i valori in ingresso sono veri, e falso in tutti gli altri casi.

- **L'operatore or (||)**, restituisce vero quando almeno un valore in ingresso è vero, e falso altrimenti.

Il C++ segue quella che viene chiamata logica cortocircuitata, secondo la quale la valutazione dell'operando di destra viene evitata quando il valore dell'operando di sinistra è sufficiente a determinare il valore dell'espressione. Ad esempio, nell'istruzione:

```
a && ((b || c) && !d)
```

la prima operazione compiuta dal calcolatore sarà valutare se a è falso: in tal caso, infatti, sarà inutile analizzare il resto dell'espressione, poiché il risultato finale dell'operazione and non potrà comunque risultare vero.

2.6 ASSEGNAMENTO ED INCREMENTI

Dell'assegnamento abbiamo già parlato nel paragrafo 2.2, ma a questo punto ne sappiamo abbastanza per comprendere il fatto che gli assegnamenti sono espressioni. Il C++, infatti, eredita dal C la filosofia secondo la quale le istruzioni possono restituire un valore (il che è quel che fanno nella maggior parte dei casi). L'assegnamento restituisce l'espressione espressa alla destra dell'uguale. Detto con un esempio, scrivere:

```
a = 5 + b
```

ha il doppio significato di "poni a uguale a 5+b" e 5 + b. Puoi verificarlo provando ad… assegnare un assegnamento:

```
int a, b;

a = b = 5;
```

Questa scrittura è perfettamente consentita proprio in virtù del fatto che l'assegnamento b = 5 è un espressione (vale 5). In questo modo, dopo l'esecuzione di queste righe, sia a che b saranno impostati a 5. Vedere l'assegnamento come espressione quindi, rende possibile scrivere assegnamenti multipli con una sola istruzione, e altri trucchi che tendano a far risparmiare righe di codice – e alcuni tendono anche ad abusarne, producendo codice illeggibile, secondo la valida massima di Orazio: "sono conciso, divento oscuro". La brevità negli assegnamenti è un chiodo piuttosto fisso per chi scrive codice C++ (e soprattutto C), tanto che sono stati introdotti degli speciali operatori (detti d'incremento), per semplificare dei casi autoreferenziali di assegnamento. Osserva, per esempio, questo caso:

```
int a = 1; // a = 1
a = a + 1; // a = 2
a = a + 1; // a = 3
```

L'operazione a = a + 1 consiste nell'incrementare a di un'unità. Questo caso è tanto frequente che è stato creato l'operatore ++. Scrivere a++, quindi, è equivalente a scrivere a = a +1; quando si vuole usare l'operatore ++ come un'espressione di assegnamento, invece, esiste una fondamentare differenza fra l'uso della notazione prefissa (++a) e quello della notazione postfissa (a++). Per capire qual è, puoi guardare questo codice:

```
//Esempio operatore ++
#include <iostream>
using namespace std;
int main() {
  int a;
  a = 0;
  cout << a++ << endl;
  a = 0;
```

```
  cout << ++a << endl;
  return 0;
}
```

Entrambi gli operatori ++, qui, hanno il significato di a = a + 1. Tuttavia l'output generato dai due cout è diverso.

```
0
1
```

Ciò accade perché l'operatore ++ prefisso restituisce prima il valore della variabile, e poi la incrementa; l'operatore postfisso, invece, fa l'inverso. Analogamente esiste la possibilità di decrementare la variabile con l'operatore -- (prefisso o postfisso). Per incrementi maggiori di un'unità e per altre operazioni autoreferenziali (aritmetiche o bit-a-bit), si usano degli simboli composti dall'operatore di partenza seguito dall'uguale, ad esempio:

```
int a = 6;
a +=  2; //a = a +  2
a %=  3;      //a = a %  3
a <<= 2;      //a = a << 2
a ^=  41;     //a = a ^  40
```

Alla fine del codice riportato qui sopra, il valore di a sarà 9. Avremmo potuto ottenere lo stesso risultato sostituendo ogni linea codice con il relativo commento.

2.7 OPERATORI BIT-A-BIT

In questa carrellata di tipologie d'espressioni siamo partiti dalle fondamentali, per poi scendere alle più specializzate. Gli operatori bit-a-bit mostrati in tabella 2.4 sono probabilmente gli operatori meno utilizzati da molti pro-

grammatori C++, mentre sono il pane quotidiano fra la ristretta cerchi di coloro che si occupano di ottimizzazione o della manipolazione di messaggi digitali. Dandogli uno sguardo, potrai vedere che gli operatori bitwise sono simili agli operatori logici; i neofiti, infatti, li confondono spesso, accorgendosene solo quando il codice non funziona secondo le loro aspettative.

Simbolo	Operazione	A	b	r
~	Complemento	0 1		1 0
&	And	0 1 1 0	0 1 0 1	0 0 0 1
|	Or	0 1 1 0	0 1 0 1	0 1 1 1
^	Xor	0 1 1 1	0 1 0 1	0 1 1 0

Tabella 2.4: Operatori logici

La differenza fondamentale che distingue gli operatori bitwise dai logici, è che mentre questi ultimi agiscono sul valore logico degli operandi, i primi applicano dei confronti su ogni singolo bit. L'operazione di complemento (~), ad esempio, restituisce un valore in cui ogni bit dell'operando viene negato. Pertanto ~120 indica il complemento del byte 120 (01111000), ovverosia 135 (10000111). Proprio questo caso è utile per indicare il fatto che le operazioni bit-a-bit sono particolarmente sensibili al tipo di dati in ingresso: quanti bit compongono l'informazione? L'espressione si deve intendere signed o unsigned? A risposte diverse corrispondono risultati completamente differenti. Gli altri tre operatori bit-a-bit si usano per molte ragioni, soprattutto nel campo della manipolazione di segnali digita-

li. Un impiego tipico è l'applicazione di una maschera al primo operando, in modo da estrarne solo la parte di informazione che interessa (il che è comune, ad esempio, per creare effetti di trasparenza nei programmi di grafica). Il seguente esempio scompone un byte nei bit costitutivi, grazie all'uso degli operatori bit-a-bit.

```
// Scomposizione di un byte.
// esempio di operatori bit-a-bit.
#include <iostream>
using namespace std;
int main() {
   cout << "scrivi il numero da convertire: ";

   //preleva il dato
   int dato;
   cin >> dato;

   //converte in byte
   char byte = (char)dato; // casting esplicito

   cout << "il valore in binario è:"
        << (bool)(byte & (1<<7))     // 10000000
        << (bool)(byte & (1<<6))     // 01000000
        << (bool)(byte & (1<<5))     // 00100000
        << (bool)(byte & (1<<4))     // 00010000
        << (bool)(byte & (1<<3))     // 00001000
        << (bool)(byte & (1<<2))     // 00000100
        << (bool)(byte & (1<<1))     // 00000010
        << (bool)(byte & (1<<0))     // 00000001
        << endl;

  return 0;
}
```

Un output di esempio di questo programma è:

```
Scrivi il numero da convertire: 254
Il byte, in binario, è: 11111110
```

Un'analisi del codice rivela innanzitutto un'operazione di casting (la vedremo nel paragrafo 2.10) per la conversione da intero a char, in modo da richiedere un numero intero all'utente (e non un carattere ASCII), ma operare su dati di 8 bit. Per ricavare il contenuto di ogni bit abbiamo creato una maschera specifica, ottenuta semplicemente un bitshift sinistro il byte 00000001 (come mostrato nei vari commenti). Una volta ottenuta questa maschera, possiamo utilizzarla per un and bitwise (&), in modo da annullare tutti i bit del byte, ad esclusione di quello nella posizione considerata. Otterremo così un byte che è uguale alla maschera se il bit è presente, ed è 0 altrimenti. Poiché qualunque numero diverso da zero corrisponde al valore logico 1, un casting a bool permette di stampare la cifra corretta

2.8 OPERATORE TERNARIO

L'ultimo operatore di cui dobbiamo discutere è molto utilizzato dai programmatori che non temono la complicazione a favore della concisione. Si tratta dell'operatore ternario (?), che introduce il principio delle espressioni condizionali. Per un esempio pratico e facilmente comprensibile, puoi riconsiderare il codice del paragrafo 2.5, ed emozionarti di fronte al nostro primo bug: cosa succede, infatti, se diamo come secondo operando il valore 0? L'applicazione va in crash, e dalla (**figura 2.1**) si apprende in maniera tanto evidente quanto drammatica che a dare problemi è l'operazione di divisione. Il problema, come avrai intuito, è che la divisione per zero è un'operazione illegale, che genera un'eccezione: poiché il programma non è in grado né di prevenirla, né di gestirla, il risultato è il crash dell'applicazione.

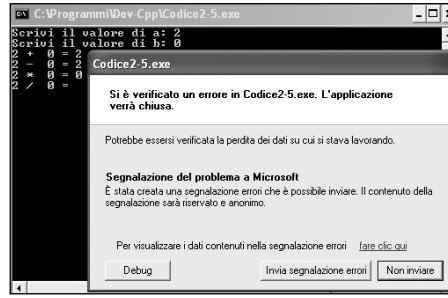

Figura 2.1: Crash dell'applicazione a causa della divisione

Impareremo a gestire le eccezioni più avanti, per ora ci accontenteremo di prevenirla – ovverosia di fare in modo che il programma non si trovi mai a dover eseguire una divisione per zero. Le vie che possono condurre a questo risultato sono diverse: dalla meno sicura (richiedere all'utente di non porre a zero il valore b), all'uso di un if (struttura che spiegheremo nel prossimo capitolo). Noi utilizzeremo un'espressione condizionale, che restituisca il risultato di a/b se b è diverso da zero, e 0 in caso contrario; anche se questo valore non è corretto, così facendo previeniamo l'operazione pericolosa, evitando il crash dell'applicazione (in C++ questa è comunque una pratica da rifuggire, come vedremo più avanti).. Un'espressione condizionale usa l'operatore ternario, secondo la seguente sintassi:

```
condizione ? se_vero : se_falso
```

Se il valore dell'espressione booleana condizione è true, l'espressione assumerà il valore contenuto in se_vero, altrimenti sarà equivalente a se_falso. Nel nostro caso, la riga incriminata si trasforma così:

```
cout << a << " /  " << b << " = " << (b ? a/b : 0) << endl;
```

L'espressione condizionale si traduce come: se b è diverso da 0, allora scrivi a/b, altrimenti scrivi 0. La medesima operazione dev'essere effettuata per il modulo.

2.9 CASTING

Nel paragrafo 2.8 abbiamo avuto la necessità di convertire un valore int in un char. Operazioni di questo tipo prendono il nome di cast, e possono essere implicite oppure esplicite. I casting impliciti avvengono spesso, e a volte senza che il programmatore se ne renda conto: un esempio di casting implicito sono le chiamate a cout, che convertono le espressioni dal loro tipo originario al tipo di stream su cui effettuare l'inserimento. Questi casting funzionano da soli, e non causano problemi, a patto di conoscere ciò che avviene dietro le scene. In altri casi, è necessario rendere i casting espliciti: questo dev'essere fatto quando il casting non è ovvio, o quando c'è rischio di perdita di precisione, ad esempio in una conversione da range maggiore (int) a range minore (char). In questi e in altri casi, il casting può essere imposto mettendo tra parentesi tonde il tipo di dato in cui si vuole convertire il valore.

```
//Esempio di casting

#include <iostream>
using namespace std;
int main() {
unsigned char a = 140;
cout << "il byte " << (int)a << "corrisponde al simbolo " << a;
}
```

Il codice qui sopra riportato è un esempio di un caso di ambiguità: se passiamo a cout il valore a, questo ne stamperà a video il carattere ASCII. Se, invece, siamo interessati al valore numerico del byte a, dobbiamo eseguire il casting di tipo (int)a. Risultato dell'esecuzione:

```
il byte 140 corrisponde al simbolo î
```

2.10 ESERCIZI

- Estendi l'applicazione del calcolatore per la valutazione degli operatori bit-a-bit e delle espressioni logiche.

- Il C++ non prevede un operatore logico di tipo xor (qualcosa come ^^). Sapresti indicare un espressione equivalente?
  ◦ Suggerimento: Perché scrivere a != b non è sufficiente?
- In un videogioco che ha bisogno di prestazioni elevate viene richiesto di verificare se un punto generato casualmente si trova alle coordinate (x=15; y=20). Come creeresti l'espressione di valutazione?

- Estensione: È preferibile scrivere (*x==15 && y==20*) oppure (*y==20 && x==15*)? o è indifferente?
  ◦ Suggerimento: non è del tutto indifferente. Pensa alle risoluzioni video tipiche, alle prestazioni e alla logica cortocircuitata.

- Scrivi un'espressione che, date in ingresso due espressioni a e b, restituisca "Maggiore" se a è maggiore di b, "Minore" se è minore e uguale altrimenti.

  ◦ Suggerimento: dovrai usare l'operatore ternario più di una volta.

- Un vecchio trucco dell'informatica è lo xor swap, ovvero la possibilità di scambiare il contenuto di due variabili intere senza usare variabili temporanee, secondo il seguente codice:

```
a ^= b;
b ^= a;
a ^= b;
```

- Prova a seguire l'esecuzione di quest'algoritmo su alcuni esempi numerici, e a spiegare come funziona.

CONTROLLO DEL FLUSSO

Nel precedente capitolo abbiamo visto i tipi di dato fondamentali e la loro combinazione in espressioni. Abbiamo anche cominciato a notare come una programmazione con un flusso d'istruzioni lineare non sia uno strumento sufficiente (il caso della divisione per zero), e come l'esecuzione di compiti relativamente semplici possa tramutarsi in un numero troppo elevato di istruzioni (convertire un numero a 64bit in binario, ad esempio). Per questi ed altri scopi, i linguaggi di programmazione come il C++ prevedono l'utilizzo di costrutti che alterino il flusso di esecuzione, facendolo tornare indietro, saltare in avanti (in rari casi), e ramificare. Per illustrare tutto ciò graficamente, faremo uso dei **diagrammi di flusso**. Quello presentato nel (**diagramma 3.1**) rappresenta il flusso lineare che abbiamo usato finora.

Diagramma 3.1: Costruttori di selezione.

3.1 COSTRUTTI DI SELEZIONE

3.1.1 IF

Il caso della divisione per zero può essere risolto eseguendo l'istruzione pericolosa solo se il secondo operando è diverso da zero. Una struttura che permette questa operazione prende il nome di costrutto di selezione, e nella sua forma più semplice (**diagramma 3.2**) ha la seguente sintassi:

```
if (condizione)
istruzioni
```

Diagramma 3.2: If.

Laddove il termine "istruzioni" può indicare una singola istruzione o un gruppo d'istruzioni (in questo caso, è obbligatorio racchiuderle fra parentesi graffe), che saranno eseguite soltanto se l'espressione booleana espressa in condizione è vera. Il caso della divisione per zero può quindi essere risolto facilmente così:

```
if (b) //oppure if(b != 0)
cout << a << " /  " << b << " = " << a / b << endl;
```

incorporando anche l'istruzione del modulo, è necessario usare le parentesi graffe:

```
if (b) //oppure if(b != 0)
{

cout << a << " /  " << b << " = " << a / b << endl;
cout << a << " %  " << b << " = " << a % b << endl;
}
```

In questo modo le operazioni di divisione e modulo verranno eseguite soltanto se b è diverso da zero, mentre verranno saltate in caso contrario. Questo potrebbe essere un comportamento accettabile o meno, a seconda degli intenti. Potremmo alterare il codice in maniera da prevedere anche la gestione del caso contrario (**diagramma 3.3**).
Questo è possibile per mezzo di un'estensione della sintassi:

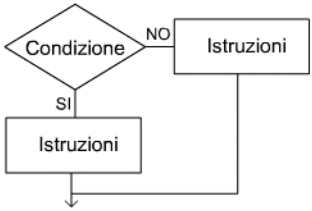

Diagramma 3.3: If..else

```
if (condizione)
istruzioni
else
istruzioni
```

Le istruzioni previste nel blocco else saranno eseguite solo nel caso in cui condizione sia false. Da ciò possiamo ricavare il codice che ci serve:

```
if (b)
cout << a << " /  " << b << " = " << a / b << endl
        << a << " %  " << b << " = " << a % b << endl;
else
cout << a << " /  " << b << " = " << "Impossibile" << endl
    << a << " %  " << b << " = " << "Impossibile" << endl;
```

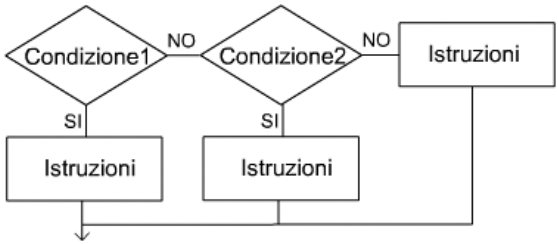

Diagramma 3.4: if..else if…else

Gli if, così come ogni altro tipo di costrutto, possono essere nidificati. È, cioè, possibile inserire una struttura if all'interno del blocco di istruzioni. Un esempio molto comune nella programmazione è l'estensione dell'if a più casi alternativi.

```
if (condizione1)
    istruzioni
else if (condizione2)
    istruzioni
else
    istruzioni
```

Come si vede dal (**diagramma 3.4**), questo significa estendere "verso destra" indefinitamente il flusso dell'esecuzione, fino all'eventuale else generico.

3.1.2 SWITCH

L'uso degli if mostrato nel diagramma 3.4 assume dimensioni inquietanti quando i casi da considerare sono un numero non banale, perché presume che per ognuno di essi occorra nidificare l'esecuzione: arrivare ad innestare dieci o quindici if l'uno dentro l'altro è contro ogni principio della buona programmazione. In simili casi, ci si può avvalere del costrutto switch, che ha la seguente sintassi:

```
switch(discriminante) {
case n1:
    istruzioni
case n2:
    istruzioni
default:
    istruzioni
};
```

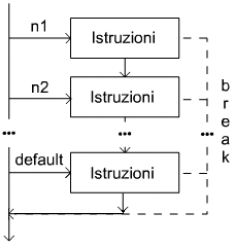

Diagramma 3.5: if..else if…else

laddove discriminante è un intero (o un'enumerazione – vedi paragrafo 4.1.2) che può assumere uno dei valori previsti nei vari case. Se non esiste alcun caso previsto per il valore di discriminante, il controllo passerà alle istruzioni previste in default (se questo è stato previsto). Occorre fare attenzione all'uso del costrutto switch, coscienti del fatto che (come mostra il diagramma 3.5) il codice è in realtà un blocco continuo che non si interrompe alla fine di un case, ma prosegue nell'esecuzione di tutti quelli che seguono – il che spesso non corrisponde al comportamento desiderato. Per questo, solitamente si aggiunge una direttiva break (vedi paragrafo 3.3.1) alla fine di ogni caso, per uscire dal blocco.

```
switch(lati) {
case 3:
    cout << "La figura è un triangolo";
    break;
case 4:
cout << "La figura è un quadrato";
break;
case 5:
cout << "La figura è un pentagono";
    break;
default:
    cout << "La figura è un poligono";
```

```
    break;
}
```

Se in quest'esempio ci fossimo dimenticati di inserire le tre istruzioni break alla fine di ogni caso, il programma avrebbe stampato il messaggio relativo alla figura giusta, più tutti quelli dei casi successivi.

3.2 COSTRUTTI D'ITERAZIONE

I costrutti di iterazione (o cicli) sono fondamentali in tutti quei casi in cui sia necessario ripetere più volte la stessa sequenza d'istruzioni. Le calcolatrici, ad esempio, non terminano la propria esecuzione dopo aver fornito il risultato, ma continuano ad attendere nuovi input.

3.2.1 WHILE

La struttura while permette di ripetere una sequenza d'istruzioni fintantoché una condizione è vera. La condizione può essere verificata prima dell'esecuzione (**diagramma 3.6**), secondo la sintassi:

```
while(condizione)
    istruzioni
```

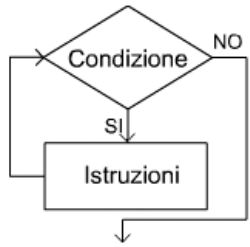

Diagramma 3.6: While

oppure dopo (**diagramma 3.7**) secondo l'alternativa:

```
do
istruzioni
while(condizione);
```

Diagramma 3.7: Do..While.

Anche i costrutti while possono essere nidificati a piacimento, come dimostra la seguente variazione (l'ultima!) della calcolatrice (vedi **diagramma 3.8**).

Diagramma 3.8: Calcolatrice minima.

```
//Calcolatrice minima (versione finale)
#include <iostream>

using namespace std;

int main() {
   char risposta;
```

```
  do {
    int a, b;

    //richiesta dei due numeri

    cout << "Scrivi il valore di a: ";
    cin >> a;
    cout << "Scrivi il valore di b: ";
    cin >> b;

    //stampa dei risultati

    cout << a << " +  " << b << " = " << a + b    << endl
         << a << " -  " << b << " = " << a - b    << endl
         << a << " *  " << b << " = " << a * b    << endl
         << a << " << " << b << " = " << (a << b) << endl
         << a << " >> " << b << " = " << (a >> b) << endl;

    if (b)
      cout << a << " /  " << b << " = " << a / b << endl
           << a << " %  " << b << " = " << a % b << endl;
    else
      cout << a << " /  " << b << " = " << "Impossibile" << endl
           << a << " %  " << b << " = " << "Impossibile" << endl;

    do {
      cout << endl << "Vuoi continuare [S/N]";
      cin >> risposta;
    } while(risposta != 'S' && risposta != 'N');
  } while (risposta == 'S');

  return 0;
}
```

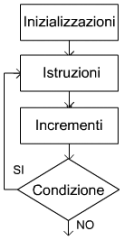

Diagramma 3.9: For.

3.2.2 FOR

Il ciclo for è senz'altro il costrutto d'iterazione più complesso e più usato nella programmazione, e si utilizza in quei casi in cui è necessario ripetere l'esecuzione di un blocco d'istruzioni tenendo traccia del numero di repliche, mediante l'incremento del valore di un contatore o del riferimento di un iteratore. La sintassi è:

```
for (inizializzazioni; condizione; incrementi)
    istruzioni
```

Può essere utile seguire il diagramma 3.9, per capire esattamente l'ordine in cui vengono eseguite le varie parti del ciclo. Le inizializzazioni consistono in una dichiarazione/assegnamento (o più di una, separate da virgole), e si usano per porre il contatore ad un valore iniziale. Gli incrementi consistono in una istruzione (o più, separate da virgole), che aumenti il valore del contatore. Possiamo dare un esempio di ciclo for, rendendo molto più compatto il convertitore binario dell'esempio 2.7:

```
// Scomposizione di un byte mediante ciclo for.

#include <iostream>

using namespace std;
```

```
int main() {
   cout << "scrivi il numero da convertire: ";

   //preleva il dato
   int dato;
   cin >> dato;

   //converte in byte
   char byte = (char)dato; // casting esplicito

   cout << "il valore in binario è:";

   for (int i=7; i>=0; i--)
       cout << (bool)(byte & (1 << i))

   cout << endl;

  return 0;
}
```

In questo caso il ciclo for viene utilizzato per un conteggio alla rovescia, che termina dopo l'iterazione (i==0). Molto utilizzato è anche il conteggio di tipo opposto:

```
for (int i=inizio; i<fine; i++)
```

che esegue il ciclo partendo dal valore inizio e fermandosi al valore (fine-1)

3.3 SALTI

Oltre ai costrutti visti fin qui, il C++ prevede tre parole chiave per l'alterazione del flusso d'esecuzione. Questi sono break, continue e goto.

3.3.1 BREAK

La parola chiave break può essere usata solo all'interno di un blocco (if, switch, while, for), per forzarne l'uscita. Le istruzioni successive alla chiamata non verranno eseguite. Ad esempio:

```
for(int i=0; i<10; i++) {
    if (i==6)
        break;

    cout << i;
}
```

In questo esempio, il ciclo che stamperebbe le dieci cifre 0123456789, in realtà produrrà il solo 012345, dopodiché verrà interrotto. Come abbiamo visto nel paragrafo 3.1.2, l'istruzione break viene usata frequentemente nei costrutti switch.

3.3.2 CONTINUE

La parola chiave continue è per molti versi analoga a break. Si utilizza sempre all'interno di blocchi e altera il flusso dell'esecuzione, ma, a differenza di break, non esce dal ciclo. Si limita, invece, a saltare il resto dell'iterazione corrente, richiamando subito la successiva. Modificando l'esempio precedente con:

```
for(int i=0; i<10; i++) {
    if (i==6)
        continue;

    cout << i; //verrà saltata per i==6
}
```

l'output generato sarà 012345789. Quando i sarà uguale a 6, in-

fatti, l'istruzione continue interromperà l'iterazione, riprendendo il ciclo da i=7.

3.3.3 GOTO

L'istruzione goto viene usata per eseguire un salto incondizionato all'interno della funzione corrente, verso una riga che sia stata contrassegnata da una determinata etichetta. Un'etichetta viene specificata anteponendo ad un'istruzione un identificativo seguito dai due punti (come nei case del costrutto switch, il quale infatti usa delle etichette ed ha un funzionamento intimamente connesso al goto).
Ecco un semplice esempio dell'uso di goto:

```
#include <iostream>
using namespace std

int main() {
goto uscita;
cout << "Questa riga non verrà mai eseguita!";
uscita:
    return 0;
}
```

L'uso del goto è considerato una pessima pratica, dai tempi di uno storico articolo di Dijkstra [4], perfettamente riassunto dal suo stesso incipit: "La qualità dei programmatori è una funzione decrescente della densità delle istruzioni goto nei programmi che producono." La ragione di un simile giudizio è che abusando del goto si arriva facilmente a creare codice illeggibile (spaghetti code), senza alcuna necessità reale. Non esistono casi, infatti, in cui l'uso del goto non sia sostituibile con delle perifrasi o semplicemente scrivendo in maniera più strutturata il proprio codice. Il C++ lascia comunque la possibilità di usare tale istruzione, per quei programmatori più esperti che

sanno quando infrangere il tabù: ad esempio per uscire dai cicli più esterni senza dover usare delle scomode variabili temporanee.

3.4 VISIBILITÀ

I costrutti if, while e for prevedono nella loro sintassi la dicitura istruzioni. Questa permette, come abbiamo già visto, due possibilità:

• Una sola istruzione: Ad esempio:

```
if (numeroLati == 4)
    area = lato * lato;
```

• Un blocco che racchiuda più istruzioni. Ad esempio:

```
if (numeroLati == 4) {
    area        = lato * lato;
    perimetro   = 4 * lato;
}
```


Un blocco, quindi, si caratterizza per la presenza delle parentesi graffe. Qui analizzeremo il fatto che le variabili dichiarate all'interno di un blocco godono di una visibilità minore rispetto a quelle esterne. Per capire il concetto della visibilità (o scope), occorre capire come si svolge il ciclo di vita delle variabili: nel momento in cui una variabile viene dichiarata, viene allocato lo spazio necessario nello stack, e la sua locazione di memoria diventa accessibile. In seguito tale variabile può essere inizializzata, assegnata e ridefinita. Infine, non appena questa esce dal raggio di visibilità, viene distrutta. La visibilità di una variabile si esaurisce all'uscita dal blocco in cui questa si trova. In questo esempio:

```
int main() {
```

```
    int n = 0;

    while(n < 20) {
        int i=1;
        n += i*2;
    }

    n = i;     // errore! i non è più visibile
    return 0;
}
```

la variabile n diventa visibile all'inizio della funzione, e scompare alla fine, pertanto può essere usata all'interno del ciclo while senza problemi. L'errore, invece, riguarda l'uso della variabile i, che viene dichiarata all'interno del ciclo while, e pertanto perde visibilità non appena il ciclo finisce. Questo spiega perché il compilatore segnala la riga n = i con un errore di tipo: "variabile non dichiarata". In un blocco si può anche dichiarare una variabile già esistente all'esterno. In tal caso il riferimento a tale variabile sarà ridefinito, e non sarà più possibile fare riferimento alla variabile esterna fino alla fine del blocco:

```
#include <iostream>
using namespace std

int main() {
    int n = 0;

    {    //blocco interno
        int n = 2;
        cout << "all'interno del blocco, n vale " << n << endl;
    }
```

```
    cout << "all'esterno del blocco, n vale " << n << endl;

    return 0;
}
```

Il risultato dell'esecuzione sarà:

```
all'interno del blocco, n vale 2
all'esterno del blocco, n vale 0
```

L'esempio mostra chiaramente che si tratta di due variabili completamente diverse che condividono soltanto lo stesso nome; inoltre introduce il concetto che i blocchi possono essere usati anche senza un particolare costrutto for, if o while, che li preceda - anche se questo, nella pratica comune, avviene molto raramente.

3.5 ESERCIZI

- Scrivi un algoritmo per il calcolo del fattoriale
- Scrivi un algoritmo per l'elevamento a potenza.
- Prova a scrivere un ciclo infinito attraverso un costrutto while.
- Prova a scrivere un ciclo infinito attraverso un costrutto for.
- Scrivi un programma che sia in grado di stampare a video la tavola pitagorica 10x10.
  - Suggerimento: dovrai nidificare due cicli for

TIPI AVANZATI

In questo capitolo vedremo come manipolare i dati fondamentali, per creare dei tipi di dato più complessi (strutture), per creare sequenze di più elementi (array), e vedremo come questi dati vengano allocati in memoria. Alla fine di questo capitolo saremo quindi in grado di gestire tutte le strutture che sono comunemente adoperate nella programmazione di stampo procedurale. Se sei un novizio della programmazione, o non hai comunque un background solido su strutture ed algoritmi di base (liste collegate, alberi binari, ordinamento, etc…) ti consiglio fortemente la lettura di [5] o equivalenti.

4.1 COSTANTI, ENUMERAZIONI E TYPEDEF

Il C++ offre degli strumenti per migliorare la leggibilità e la manutenibilità del codice, in quei casi in cui alcuni valori siano noti indipendentemente dall'esecuzione: le costanti e le enumerazioni. Per semplificare la ridefinizione dei tipi, invece, il C++ permette di usare la parola chiave typedef.

4.1.1 COSTANTI

Alcuni valori non cambiano mai durante il corso dell'esecuzione, talvolta perché sono delle costanti per loro natura (ad esempio, pi greco), altre volte perché il programmatore non vuole che il loro valore possa cambiare in seguito all'inizializzazione. A tal fine, si può anteporre ad una dichiarazione la parola chiave const, che indica una limitazione di tipo per un valore, che rimarrà costante per tutto l'arco di visibilità.

```
int main() {
    const int PI = 3.14;

    double angoloRetto    = PI/2;
```

```
    double angoloGiro      = PI*2;

    return 0;
}
```

Così facendo si ottiene una scrittura molto più manutenibile e leggibile per il programmatore, senza introdurre alcun overhead: un compilatore appena sopra la soglia della decenza, infatti, sarà comunque in grado di risolvere la costante per sostituzione, rendendo così questa scrittura identica a quella ottenuta utilizzando direttamente il numero.

4.1.2 ENUMERAZIONI

I programmatori C++ spesso fanno più uso delle enumerazioni, che delle costanti, in tutti quei casi in cui un tipo di variabile può assumere soltanto degli stati precisi. Un esempio di enumerazione è:

```
enum LuceSemaforo {
    ROSSO,
    GIALLO,
    VERDE
}
```

e la sintassi esatta di enum:

```
enum nome {
    STATO1 = valore,
    STATO2 = valore,
    …
};
```

Laddove "= valore" è opzionale, e indica la possibilità di stabilire un valore intero per lo stato. Se tale inizializzazione non viene fatta,

si userà un ordine crescente (partendo da zero, o dal primo valore inserito). Un'enumerazione si usa come un intero qualsiasi, ed è quindi particolarmente utile per rendere più leggibili gli switch. Ad esempio, avendo una variabile luce di tipo LuceSemaforo, possiamo scrivere:

```
switch(luce) {

case ROSSO:
    fermati();
    break;
case GIALLO;
    preparati();
    break;
case VERDE:
    vai();
    break;
}
```

4.1.3 TYPEDEF

Typedef permette un meccanismo simile a quello delle costanti, ma agisce sui tipi, anziché sui valori. Quando un tipo di dato si usa molto, è possibile ridefinirlo al fine di migliorare la leggibilità del codice: un operazione che, comunque, viene realizzata solo dai più esperti, quando vogliono usare i tipi in maniera più coerente col framework che stanno adoperando (o costruendo):
Nella programmazione windows (vedi il file header windef.h), ad esempio, vengono definiti molti typedef simili a questo:

```
typedef unsigned long DWORD;
```

Questo permette al programmatore una dichiarazione di questo tipo:

```
DWORD numero;
```

che sarà del tutto equivalente a scrivere:

```
unsigned long numero;
```

Se sei un novizio nel C++, probabilmente non avrai nessun bisogno di usare i typedef; ma se, invece, fai largo uso di tipi complessi, il typedef può essere una grande comodità.

4.2 STRUTTURE E UNIONS

Enumerazioni e typedef permettono al programmatore di inventare nuovi tipi di dato a partire dai primitivi. La libertà concessa, però, è molto poca: le enumerazioni saranno sempre interi, e i typedef non possono in alcun modo essere considerate delle aggiunte, bensì dei sinonimi.

4.2.1 STRUTTURE

Una maniera semplice per creare tipi di dati nuovi è fornita dalla parola chiave struct, che permette di definire un tipo composto dall'insieme di più variabili, dette campi o membri. La sintassi è:

```
struct nome {
    tipo1 campo1;
    tipo2 campo2;
    …
};
```

ed ecco un esempio di dichiarazione:

```
struct Frazione {
    int numeratore;
```

```
    int denominatore;
};
```

Una volta che una struttura è stata definita, è possibile trattarla esattamente come un tipo di dati primitivo. Per accedere ai campi si usa l'operatore punto (.).

```
int main() {
    struct Frazione {
        int numeratore;
        int denominatore;
};

    //creiamo una nuova f
    Frazione f;

//poniamo f a 3/5.
    f.numeratore   = 3;
    f.denominatore = 5;

    return 0;
}
```


4.2.2 UNIONI

Strette parenti delle strutture sono le unioni, definite mediante la parola chiave union. Lo scopo delle unioni è quello di creare variabili capaci di assumere tipi diversi, cercando di consumare meno memoria possibile.

Ciò viene realizzato assegnando i diversi campi costituenti l'unione allo stesso indirizzo di memoria. Qui di seguito viene definito un esempio semplificato del comportamento delle variabili di tipo Variant che vengono adottate da vari linguaggi di programmazione nell'automazione OLE.

```
union Variant {
    int valoreIntero;
    char valoreChar;
    double valoreDouble;
};
```

Le union sono una forma avanzata di ottimizzazione e non dovrebbero mai essere usate alla leggera (il mio spassionato consiglio è di evitarle), facendovi ricorso solo quando si ha una precisa necessità (ad esempio, nel costruire un ambiente run-time per un linguaggio di scripting), e un effettivo bisogno di ridurre l'impiego di memoria.

4.3 VARIABILI E MEMORIA STATICA

Nel momento in cui una variabile viene dichiarata localmente, come ad esempio in:

```
int main()
{
    int i=0;
    return 0;
}
```

il calcolatore le riserva lo spazio richiesto dal tipo (in questo caso, supponiamo 4 byte) all'interno di un blocco di memoria chiamato stack. Da quel momento in poi, i sarà associato a quel particolare indirizzo Occorre quindi fare una precisa distinzione fra indirizzo e valore. Nel seguente codice:

```
int main()
{

    int i=13, j=13;
```

```
    return 0;
}
```

i e j hanno lo stesso valore, ma diverso indirizzo. Questo può essere verificato per mezzo dell'operatore &, che restituisce l'indirizzo di una variabile. Ad esempio:

```
#include <iostream>
int main()
{
int i = 0xAABBCCDD; // in decimale, 2864434397
int j = 0x11223344; // in decimale, 287454020
    std::cout << "indirizzo di i = " << &i << ", "
              << "indirizzo di j = " << &j << ". ";
    return 0;
}
```

L'output di un esecuzione tipica può essere:

```
indirizzo di i = 0x22ff74, indirizzo di j = 0x22ff70.
```

È evidente che in questo caso un int prende quattro bytes, e le due variabili sono state memorizzate in spazi contigui. La figura 4.1 mostra lo stato di questa porzione di stack, laddove i due valori in grigio chiaro agli indirizzi (74 e 70), corrispondono ai due indirizzi puntati dalle variabili. Quando la variabile esce dall'area di visibilità, viene richiamato l'eventuale distruttore (se è un oggetto), e l'indirizzo di memoria relativo viene considerato nuovamente accessibile.

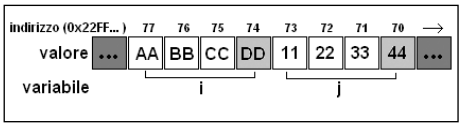

Figura 4.1: Rappresentazione dei valori sullo stack.

4.4 PUNTATORI

La nostra recente "scoperta" che le variabili non sono altro che nomi associati ad un indirizzo, ci porta a poter comprendere uno degli strumenti più utili e insidiosi a nostra disposizione, croce e delizia dei programmatori C++: i puntatori.
Un puntatore è una variabile che memorizza un indirizzo e permette di leggere il valore relativo.

4.4.1 COME SI DICHIARANO I PUNTATORI

I puntatori devono essere tipizzati (vedremo presto la ragione), pertanto la loro dichiarazione si effettua dichiarando il tipo della variabile da puntare, seguito da un asterisco. Ad esempio con:

```
char* p;
```

Dichiariamo la variabile p come puntatore a una variabile char. In seguito possiamo associare tale puntatore ad un indirizzo specifico:

```
int main()
{
    char* p;
    char  x;

    p = &x; // giusto
    p = x;  // sbagliato ("x" indica il valore, non l'indirizzo)

    p = 0x22FF74;       // sbagliato (non è un char!)
    p = (char*)0x22F74; // giusto, ma probabilmente insensato

    return 0;
}
```

Come si evince dall'esempio, la via più corretta per usare un punta-

tore è associarlo all'indirizzo di una variabile del suo stesso tipo. Con gli opportuni cast (vedi esempio in 4.4.4), è anche possibile associare un puntatore ad una variabile di tipo diverso o ad un qualsiasi indirizzo, ma questo ha solitamente poco senso (se non per quelle rare applicazioni che conoscono in anticipo l'ubicazione di una certa variabile in memoria). Quando un puntatore non punta ancora da nessuna parte, la situazione è pericolosa, perché se il programma lo utilizzasse, farebbe riferimento ad una zona di memoria non conosciuta, con esiti drammatici. Per questo un puntatore dovrebbe sempre essere inizializzato su un riferimento valido, oppure sul valore 0. In questo modo si potrà verificare facilmente se il puntatore è valido o meno:

```
#include <iostream>

int main()
{
    char* p = 0; //puntatore messo "a terra"

    if (p) //se il puntatore è valido
        std::cout << "p è valido!";

    return 0;
}
```

Poiché p non è un puntatore valido, l'istruzione "cout" (da sostituire mentalmente con un'istruzione pericolosissima che faccia accesso al valore di p) non sarà eseguita.

4.4.2 L'OPERATORE *

L'operatore di deferenziazione '*' (chiamato star (stella), per assonanza con stored (contenuto)) permette di recuperare o assegnare il valore associato a un puntatore.

```
#include <iostream>
int main()
{
    int* p;
    int  x;
p = &x;  // ora p punta a x
*p = 10; // cambio il valore della variabile puntata in 10
std::cout << "Il valore di x e' " << x;
    return 0;
}
```

L'output di questo codice, ovverosia

```
Il valore di x e' 10
```

non dovrebbe sorprenderti se hai capito quanto discusso in 3.7.2 (in caso contrario, prova a rileggerlo e a fare qualche prova pratica).

4.4.3 ACCESSO AI MEMBRI DI UNA STRUTTURA

In base a quanto detto finora, si potrebbe accedere ai membri di una struttura referenziata da un puntatore in questo modo:

```
Frazione  elemento;
Frazione*      puntatore;
(*puntatore).numeratore   = 5
(*puntatore).denominatore = 10
```

Oltre al fatto che sono necessarie le parentesi (l'operatore punto ha la precedenza sullo star), la cosa non è affatto comoda, soprattutto quando si punta ad un membro che è a sua volta un puntatore a una struttura (e si potrebbe andare avanti all'infinito). Per questo il C++ prevede un operatore particolare per indicare direttamente un

membro di una variabile referenziata da un puntatore: la freccia (->). L'esempio può così essere riscritto:

```
Frazione elemento;
Frazione*      puntatore;
puntatore->numeratore   = 5
puntatore->denominatore = 10
```

Le due scritture sono assolutamente equivalenti, quindi troverete difficilmente un programmatore C++ tanto perverso da complicarsi la vita con uno stile simile al primo esempio.

4.4.4 ARITMETICA DEI PUNTATORI

L'aritmetica dei puntatori è la ragione fondamentale per la quale queste variabili sono tipizzate. Operazioni comuni sono l'incremento e il decremento di un puntatore per permettere di accedere alle celle contigue, o la sottrazione di due puntatori per misurarne la distanza relativa. Azioni che assumeranno un senso molto più compiuto solo quando parleremo degli array; per ora puoi comunque osservare questo codice:

```
#include <iostream>

using namespace std;

int main()
{
   int i = 0xAABBCCDD;
   int j = 0x11223344;

   cout << "L'indirizzo di i e': " << &i << endl;
   cout << "L'indirizzo di j e': " << &j << endl;
   //casting che permette di puntare un singolo byte
```

```
    unsigned char *p = (unsigned char*)&j;

    //ciclo per otto elementi della memoria
    for (int a = 0; a < 8; a++, p++)
          cout << "Indirizzo: " << (int*)p << ", "
  << "valore: "    << (int)*p << endl;

    return 0;
}
```

Il risultato dell'esecuzione, in accordo con la situazione illustrata dalla (**figura 3.1**) è:

```
L'indirizzo di i e': 0x22ff74
L'indirizzo di j e': 0x22ff70
Indirizzo: 0x22ff70, valore: 68  (cioè 0x44)
Indirizzo: 0x22ff71, valore: 51  (cioè 0x33)
Indirizzo: 0x22ff72, valore: 34  (cioè 0x22)
Indirizzo: 0x22ff73, valore: 17  (cioè 0x11)
Indirizzo: 0x22ff74, valore: 221 (cioè 0xDD)
Indirizzo: 0x22ff75, valore: 204 (cioè 0xCC)
Indirizzo: 0x22ff76, valore: 187 (cioè 0xBB)
Indirizzo: 0x22ff77, valore: 170 (cioè 0xAA)
```

Voglio tranquillizzarti: comprendere questo codice richiede tempo, un confronto con la figura 4.1, e una comprensione pratica dei meccanismi dei casting che probabilmente devi ancora sviluppare. Ragion per cui, se stai ancora strabuzzando gli occhi davanti al tuo programma, non devi preoccuparti. Potrai studiare quest'esempio con calma, o tornarci più avanti. Per ora, comunque, l'output generato dall'esecuzione di questo codice sulla tua macchina può servirti per vedere come il tuo compilatore e la tua architettura gestiscono le variabili in memoria stack.

4.4.5 PUNTATORI A PUNTATORI

Un puntatore può anche puntare… a un altro puntatore! Il C++ gestisce la cosa in maniera logica e coerente, trattando il puntatore come un valore qualsiasi:

```
int n = 0, m = 1;
int*  pn   = n;   //pn punta ad n
int**       ppn  = pn;  //ppn punta a pn che punta ad n
int*** pppn = ppn; //etc…
```

Se ti stai (giustamente) chiedendo perché mai bisognerebbe puntare ad un altro puntatore, sappi che non si tratta di un simpatico gioco di società. Simili strutture vengono impiegate molto spesso – vedi paragrafo 4.6.5.

4.4.6 A COSA SERVONO I PUNTATORI?

Ho dei vaghi ricordi dei miei inizi sulla via del C, ma uno è molto forte: continuavo a chiedermi a cosa diavolo servissero i puntatori. In fin dei conti si vive anche senza! Ovviamente, non è vero: l'uso dei puntatori diventa sempre più una necessità, via via che la conoscenza progredisce, e con essa le esigenze da soddisfare. Alcuni degli usi più comuni dei puntatori comprendono:

- Accesso rapido ad un elemento di una variabile o di una collezione.
- Passaggio di valori ad una funzione per riferimento.
- Aritmetica dei puntatori associata alla navigazione negli array.
- Creazione di alberi e liste collegate.
- Gestione della memoria dinamica.

Poiché è facile ridursi a puntare a strutture non più esistenti (dangling pointer), o dimenticarsi di deallocare le strutture che abbiamo crea-

to (memory leak), causando così il crash dell'applicazione, il C++ si dà da fare per minimizzarne l'uso dei puntatori, mediante l'introduzione dei riferimenti (references) e della libreria standard, che solleva il programmatore dal dover gestire il pericoloso "dietro le quinte" di molte strutture comunemente usate.

4.5 RIFERIMENTI

I riferimenti (o references) sono simili ai puntatori, ad eccezione del fatto che usandoli non si possono più eseguire le operazioni aritmetiche sugli indirizzi per fare i consueti giochetti interessanti – ma pericolosi – che questi ultimi permettono.

4.5.1 COME SI DICHIARANO LE REFERENCES

I riferimenti si dichiarano ponendo il simbolo & (che in questo caso non ha nessuna attinenza con l'operatore indirizzo) al tipo di dato, ed è obbligatorio inizializzarli subito su una costante o una variabile:

```
#include <iostream>

int main()
{
   int valore = 10;

   int* puntatore   = &valore;
   int& riferimento = valore;

   puntatore++;     //il puntatore punta alla cella successiva
   riferimento++;   //valore ora vale 11

   std::cout << "valore e': " << riferimento;
```

```
    return 0;
}
Valore e' 11
```

L'esempio mostra come si dichiara un riferimento ad una variabile (stessa cosa è il riferimento ad una costante), e la differenza fondamentale con i puntatori: l'aritmetica usata dal puntatore è relativa all'indirizzo, quella usata dal riferimento è associata al valore. Si può a tutti gli effetti considerare un riferimento come un altro nome per la variabile referenziata.

4.5.2 ACCESSO AI MEMBRI DI UNA STRUTTURA

Poiché non c'è differenza fra usare la vera variabile o un suo riferimento, l'accesso ai membri di una struttura associata ad un riferimento si realizza sempre per mezzo dell'operatore punto.

```
int main()
{
    Frazione f;
    Frazione& ref = f;
    f.numeratore     = 1;
    ref.denominatore = 3;
    std::cout << "il numeratore vale: " << ref.numeratore << endl;
    std::cout << "il denominatore vale: " << f.denominatore << endl;

    std::system("pause");
    return 0;
}
```

Come mostra l'esempio, riferimenti e valori possono essere usati indifferentemente, senza alcun problema. Per questo motivo il comportamento dei reference è visto come più lineare e coerente rispetto

a quello dei puntatori: non bisogna ricordarsi se è il caso di usare l'operatore freccia o il punto.

4.5.3 RIFERIMENTO AD UNA VARIABILE PUNTATA

Capita spesso di dover fare riferimento ad una variabile puntata: in tal caso va usato l'operatore di dereferenziazione:

```
int x = 5;
int* ptrx = x;
int& refx = *ptrx;
```

In quest'esempio, refx fa riferimento alla variabile x; se non avessimo fatto uso dell'asterisco, avrebbe puntato al puntatore, e non alla variabile. L'operatore * va usato per ogni puntatore, compresi this e quelli restituiti da new.

```
int& refint = *new int;
//... uso il riferimento ...
delete &refint;
```

4.5.4 PERCHÉ SI USANO LE REFERENCES

Molti programmatori C++ di formazione C ignorano completamente l'esistenza delle references, e le adoperano solo quando sono costretti a farlo (per alcune operazioni di overloading degli operatori in cui non se ne può fare a meno).
Quando si presentano i requisiti di costanza richiesti dalle references, invece, il loro uso è preferibile rispetto a quello dei puntatori, perché l'approccio ai membri è più coerente, più chiaro e meno ambiguo rispetto a quello dei puntatori.
Per questo motivo, lo standard C++ invita ad usare sempre usare le references (e non i puntatori) per indicare un parametro di una funzione passato per riferimento.

4.6 VETTORI E MATRICI

Capita spesso di dover considerare un insieme di più elementi, indicizzati o meno. Gli esempi sono infiniti: i giocatori di una squadra di calcio, i contatti di una rubrica, e così via. I vettori (o _array_) e le matrici (vettori multidimensionali) sono la via più rapida ed efficace che il C++ mette a disposizione dei suoi utenti. La conoscenza di questi strumenti nei loro diversi risvolti è un fondamento imprescindibile della formazione di un programmatore.

4.6.1 LE STRINGHE COME ARRAY

Per capire come funziona un vettore, possiamo vedere come le stringhe vengono trattate comunemente in C. Cominciamo con un codice che inizializzi e memorizzi una stringa.

```
#include <iostream>

int main()
{
    char stringa[5] = "Ciao";

    std::cout << stringa;
return 0;
}
```

La (**figura 4.2**) mostra come la stringa "Ciao" viene memorizzata nello stack (stavolta mi sono permesso di girare lo stack leggendo i byte in senso crescente: cambia solo il punto di vista.).

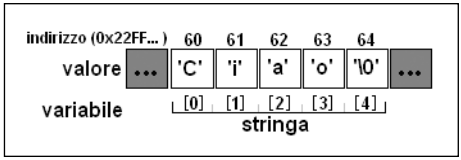

Figura 4.2: Rappresentazione sullo stack delle stringa

Dalla figura capiamo che il C++ vede le stringhe come sequenze di bytes terminate dal carattere '\0', detto carattere nullo o terminatore (byte 0). Con le nostre conoscenze in fatto di puntatori non ci sarebbe difficile associarne uno all'indirizzo di stringa, e puntare le celle ad una ad una. Tuttavia il C++ ci offre una via molto più semplice: gli array.

4.6.2 DICHIARAZIONE DI UN ARRAY

Come abbiamo visto nel codice d'esempio, la stringa è stata dichiarata come:

```
char stringa[5] = "Ciao";
```

La sintassi di dichiarazione di un array, infatti, è:

```
tipo nome[numeroElementi];
```

Il tutto è in accordo con la (**figura 4.2**), in cui risulta evidente che la stringa "ciao" è composta da cinque elementi. In questo caso, ho lasciato il numero 5 solo per semplicità ma il compilatore è abbastanza furbo da saper fare un paio di conti banali, per cui è ammessa anche la forma:

```
char stringa[] = "Ciao";
```

la quale è assolutamente equivalente. L'inizializzazione di un'array può essere effettuata anche indicando i singoli elementi fra parentesi graffe:

```
char stringa[] = {'C', 'i', 'a', 'o'};
```

Nel caso dei char questo tipo di dichiarazione è un'evidente complicazione (le stringhe letterali sono fatte apposta per questi casi). Tut-

tavia, per ogni altro tipo di dato, le parentesi graffe rimangono la soluzione più semplice, ad esempio:

```
int numeri[] = {125, 10002, 40, -20, 52};
```

4.6.3 ACCESSO AGLI ELEMENTI DI UN ARRAY

Dal momento della dichiarazione è possibile far riferimento ad uno degli elementi dell'array usando l'operatore [].

```
#include <iostream>

using namespace std;

int main()

{
char stringa[5] = "Ciao";
cout << stringa << endl;

stringa[0] = 'M';
cout << stringa << endl;

return 0;
}
```

L'output dell'esempio sarà:

```
Ciao
Miao
```

L'esempio mostra chiaramente che gli elementi si elencano da 0 in avanti. Questo, per inciso, spiega perché la maggior parte dei pro-

grammatori C++ sviluppa strani comportamenti nella vita di tutti i giorni, quando viene posta di fronte ai numeri ordinali ("Svolti alla strada[3] a sinistra. Ovverosia alla quarta". Logico, no?). Se proseguirai diligentemente nella via del C++, ti unirai presto a far parte di quest'elite sociopatologica. Fino a quel giorno, fa' molta attenzione alla numerazione degli elementi degli array cui fai riferimento.

4.6.4 GLI ARRAY COME PUNTATORI

Sicuramente non ti sarà sfuggita la strettissima connessione che lega i puntatori agli array. Anzi, a dirla tutta: un array è un puntatore al primo elemento (cioè, all'elemento zero) in esso contenuto. Puoi rendertene conto facilmente, attraverso un programma del genere:

```
#include <iostream>

int main()
{
  int array[5];

  //array punta al suo primo elemento?
  if (array == &array[0])
     std::cout << "sono uguali!";
  else
     std::cout << "sono diversi!";
  return 0;
}
```

Il responso dell'esecuzione di questo codice sarà:

```
sono uguali!
```

Il che fuga ogni dubbio in materia. Quest'equivalenza rende possibili molti giochetti, alcuni dei quali tanto utili quanto pericolosi.

Per esempio, ora sappiamo come referenziare una stringa già allocata, pertanto possiamo ritoccare l'esempio precedente in questo modo:

```
#include <iostream>
int main()
{
  int array[5];
  char* responso; //responso è una stringa! Al momento inva-lida.

  //array punta al suo primo elemento?

  responso = ((array == &array[0]) ? "uguali" : "diversi");

  //ora responso punta ad "uguali" oppure a "diversi"

  std::cout << "sono " << responso;
  return 0;
}
```

In questo caso abbiamo sfruttato l'allocazione statica delle stringhe "uguali" e "diversi" per farle puntarle direttamente da responso.

4.6.5 MATRICI E ARRAY MULTIDIMENSIONALI

Un array può estendersi su più dimensioni: quando ne occupa due si parla di matrice o di array bidimensionale, e se ne occupa di più, di array multidimensionale. Il C++ implementa il tutto in maniera molto semplice: gli array bidimensionali sono array di array, i tridimensionali sono array di array di array, e così via…Il codice che segue esempilfica la rappresentazione di una scacchiera, vista come una matrice bidimensionale di "Pezzi" che possono essere assenti, o dei tipi previsti nel gioco degli scacchi.

```
int main()
{
   enum Pezzo {
        NESSUNO,
        PEDONE,
        CAVALLO,
        ALFIERE,
        TORRE,
        RE,
        REGINA
   };
   Pezzo scacchiera[8][8];
   scacchiera[0][0] = TORRE;
   scacchiera[0][1] = CAVALLO;
   //...

   return 0;
}
```

Poiché un array è un puntatore, ne consegue che un array di array è un puntatore a puntatore. Pertanto un puntatore alla variabile scacchiera dev'essere dichiarato come:

```
Pezzo      scacchiera[8][8];
Pezzo** ptrScacchiera = scacchiera;
```

4.6.6 I PROBLEMI DEGLI ARRAY.

Non mi pare ci possano essere dubbi sul perché gli array siano fondamentali: liste e tabelle sono all'ordine del giorno in qualunque programma vada un gradino al di là della banalità più assoluta.
All'inizio del capitolo ho lodato l'efficienza di queste strutture in termini di memoria e prestazioni, qui invece parlerò delle varie e fondatissime

ragioni per cui non si dovrebbero usare gli array, se non in casi di forza maggiore. Detto in maniera concisa: sono inflessibili, incompleti, scomodi e pericolosi. Gli array sono inflessibili, perché, una volta dichiarata la loro dimensione (in maniera statica o dinamica), non c'è alcuna maniera di allargarli o restringerli, il che significa la maggior parte delle volte dover tirare a indovinare la dimensione massima degli elementi, offrendo così il fianco a quell'orda di cracker che non vedono l'ora di provocare un bug di buffer overflow nella nostra applicazione. La prassi di usare la dimensione degli array come "dimensione massima possibile" evidenzia in maniera dolorosa il fatto che queste strutture sono incomplete perché non c'è alcun modo di sapere la dimensione totale degli elementi che ne fanno parte. Tutto ciò porta alla scomodità di dover usare quasi sempre una costante per memorizzare i valori massimi, e di una variabile per tenere il conto degli elementi totali gestibili nell'array, o far ricorso a soluzioni alternative poco eleganti e poco robuste, come il carattere terminatore usato nelle stringhe. Infine, gli array sono pericolosi perché un accesso ad un loro elemento è in realtà un gioco di aritmetica dei puntatori: l'"Operazione Rischiosa" per antonomasia. Un semplice esempio: cosa succede in questo caso?


```
int main() {
    int valori[5] = {1,2,3,4,5};
    int a = valori[5];

    return 0;
}
```

In questo codice un programmatore non ancora avvezzo al fatto che gli elementi iniziano dallo zero (vedi 4.6.3), volendo richiamare il quinto elemento dell'array valori, ha in realtà richiesto il sesto, sconfinando così in una zona di memoria esterna e sconosciuta. L'operazione non causa errori di compilazione, ma al momento dell'esecuzione

l'applicazione andrà in crash nel migliore dei casi; nel peggiore, invece, potrebbe continuare a funzionare con un valore di a completamente sballato, causando potenzialmente danni ancor maggiori e più difficili da rilevare. Tutti questi problemi hanno afflitto da sempre i programmatori C.
Per questo ti suggerisco di usare i vettori solo quando lavori in un ambito noto e locale (cioè in cui il tutto si risolve in poche righe e non si trascina per l'intera applicazione), e di fare affidamento, per tutti gli altri casi, sui contenitori più gestibili forniti dalla libreria standard (ad esempio Vector). Usare il C++ è piacevole e comodo anche per questo. Il discorso vale ancor di più per le stringhe stile C, che abbiamo scoperto essere degli array in maschera: sono pericolose, scomode, inflessibili, eccetera. Usare la classe string della libreria standard, invece delle centinaia di funzioni criptiche previste dalla libreria C, è uno dei più grandi sollievi che possano essere concessi ad un programmatore dai tempi dell'invenzione della pausa caffè.

4.7 MEMORIA DINAMICA

Ci sono casi in cui si vuole creare una variabile, ma non si ha un'idea precisa sul quando la si vuole deallocare, o si vuole che perduri al di là del raggio di visibilità.

4.7.1 CREAZIONE DINAMICA DI UNA VARIABILE

In questo caso è possibile creare la variabile in maniera dinamica, facendo uso dell'operatore new, la cui sintassi è:

```
new tipo;
```

e che restituisce un puntatore all'oggetto creato.

```
main()
{
```

```
    int* x = new int;

    //usiamo x come ci pare…

    delete x;

    return 0;
}
```

Il codice mostra il tipico ciclo di vita di una variabile dichiarata dinamicamente:

- Mediante l'uso dell'operatore new, la variabile viene creata in uno spazio di memoria diverso dallo stack, chiamato heap
- Il puntatore restituito da new viene utilizzato finché si vuole
- Prima della fine del programma, viene richiamato l'operatore delete sul puntatore restituito da new, cosicché la memoria occupata dalla variabile viene liberata.

4.7.2 I PROBLEMI DELLA DICHIARAZIONE DINAMICA

Rispettare il ciclo di vita della variabile creata dinamicamente è fondamentale se si vogliono evitare alcuni dei bug più insidiosi che il C++ permetta di introdurre. Se, ad esempio, la variabile viene eliminata prima del tempo, e si fa riferimento ad essa quando già è stata cancellata dall'operatore delete, si ottiene quello che viene chiamato in gergo un dangling pointer.

```
#include <iostream>
int main()
{
```

```
    int* x = new int;
    *x = 15;

    delete x;

    std::cout << *x; //errore: x è già stato cancellato!
    return 0;
}
```

Il risultato è nel migliore dei casi un crash dell'applicazione, e nel peggiore un valore incoerente. Se, d'altro canto, ci si dimentica di cancellare una variabile creata dinamicamente si continua ad occupare dello spazio della heap inutilmente, ottenendo così quello che viene chiamato in gergo un memory leak.

```
#include <iostream>

int main()
{
    int* x = new int;
    *x = 15;

    std::cout << *x;
    return 0; //e la rimozione di x???
}
```

Queste 'falle mnemoniche' possono sembrare roba di poco conto, ma assumono ben presto proporzioni bibliche quando la creazione viene svolta in cicli e/o su array (immagina ad esempio ad un visualizzatore multimediale che ad ogni fotogramma si dimentichi di deallocare qualche Kbyte di spazio).
Gli errori di buffer overflow possono essere prevenuti con l'uso di contenitori più sofisticati, così come l'accesso ad elementi di un ar-

ray fuori dagli indici. Ma, nonostante i preziosi strumenti offerti dai vari debugger e dai tuner, gli errori di dangling pointers e memory leak sono sempre in agguato e difficili da gestire. L'insidia di questo tipo di eccezioni è che una falla può essere notata solo dopo un accumulo piuttosto ingente, e non c'è modo di stabilire con precisione da quanto un puntatore sia "a zonzo". L'unico modo di prevenire questi errori è fare molta, molta attenzione quando si allocano e deallocano variabili. L'alternativa più semplice, ma anche più dispendiosa in termini di prestazioni, è dotare il proprio C++ di un garbage collector (raccoglitore di spazzatura), che si occupi del lavoro sporco per noi [6].

4.7.3 DICHIARAZIONE DINAMICA DI ARRAYS

Negli array dichiarati staticamente, la dimensione fornita nella dichiarazione deve essere una costante. Ovverosia, non è possibile fare qualcosa del genere:

```
#include <iostream>
using namespace std;
int main()
{
    int max = 0;

    cout << "Numero di elementi dell'array: ";
    cin >> max;

    int valori[max]; //errore: max non è costante!

return 0;
}
```

La dichiarazione dinamica, invece, permette una simile operazione, grazie all'operatore new[], che crea un nuovo array sullo heap, e al

suo corrispettivo delete[] che lo rimuove.

```
#include <iostream>
using namespace std;

int main()
{
    int max = 0;

    for (int i=0; i<3; i++) {
        cout << "Numero di elementi dell'array: ";
        cin >> max;

        int* valori = new int[max]; //bene.
        delete[] valori;
    }

    return 0;
}
```

Come si vede dall'esempio, questo non solo permette di dichiarare array della dimensione desiderata, ma anche di sfruttare il ciclo allocazione/deallocazione per cambiarne la dimensione.

PARADIGMA PROCEDURALE

Finora i nostri sforzi si sono concentrati sui fondamenti del linguaggio, per quanto riguarda i dati e la gestione del flusso. Ora ne sappiamo abbastanza per guardare oltre: tutti i nostri esempi non sono mai usciti dall'interno della funzione main, ma, dai tempi del tramonto dei vecchi sistemi monolitici, questo è improponibile per ogni progetto che superi la complessità dell' "hello world".

L'architettura di tipo procedurale ha come scopo quello di spezzare il codice in blocchi distinti, possibilmente riutilizzabili, che assolvano ciascuno una funzione specifica, e contribuiscano a sorreggere il peso dell'applicazione. In questo capitolo vedremo come si scrivono e dichiarano le **funzioni**, come si organizza un progetto su più files, secondo questo stile di programmazione.

5.1 L'ALTRA FACCIA DI MAIN

Colgo l'occasione di quest'inizio capitolo per svelarti un segreto che ho taciuto finora: la funzione main ha un'altra faccia. Può essere usata in questo modo:

```
int main(int argc, char* argv[])
{
    return 0;
}
```

La differenza sta fra le parentesi, dove si trovano gli **argomenti** che vengono passati alla funzione.

Una versione di main non accetta argomenti mentre, quest'ultima, invece, ne vuole due. Questi argomenti, peraltro, sono un'accoppiata che abbiamo imparato bene a conoscere nel capitolo precedente: un array (di stringhe in stile C) chiamato argv, e il relativo numero di elementi in argc. Tali stringhe non sono altro che i **parametri** passati dall'esterno alla funzione, attraverso la shell di

comando. Possiamo provare questo codice:

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    cout << "sono stati passati " << argc << " argomenti" << endl <<
                                                                endl;

  for(int i=0; i<argc; i++)
        cout << "Argomento " << i << ": " << argv[i];

  return 0;
}
```

Se facciamo girare questo codice dall'IDE l'output sarà:

```
sono stati passati 1 argomenti
Argomento 0: C:\...\NomeProgramma.exe
```

Non sarà il massimo della grammatica, ma ci fa capire che il primo argomento è il nome stesso del programma. Puoi provare ad aggiungere dei parametri dalla linea di comando, o dall'IDE stesso, e vedere come risponde l'applicazione.

5.2 DEFINIZIONE DI FUNZIONE

Main è un esempio della funzione tipo: ha un valore di ritorno (int), e può avere degli argomenti. Possiamo provare a definire una nostra funzione sul modello dell'esercizio 3.1, per l'elevamento a potenza.

```
#include <iostream>
```

```
using namespace std;
int eleva(int base, int esponente)
{
int risultato = base;
if (base == 0)
risultato = 1;
else
for(int i=0; i<esponente-1; i++)
risultato *= base;
return risultato;
}

int main()
{
int b, e;

cout << "base:      "; cin >> b;
cout << "esponente: "; cin >> e;

cout << "il risultato e' " << eleva(b, e);

return 0;
}
```


Penso che l'esempio dimostri chiaramente come si scrive una propria funzione. La parola chiave **return** termina la funzione e restituisce al chiamante un dato. Return può quindi essere utilizzato più volte nel corso della funzione, in corrispondenza dei punti d'uscita. Per dimostrarlo, ecco una versione alternativa (ma equivalente) della funzione di elevamento a potenza:

```
int eleva(int base, int esponente)
{
```

```
if (base == 0)
return 1;

int risultato = base;

for(int i=0; i<esponente-1; i++)
risultato *= base;

return risultato;
}
```

Decidi tu quale delle due ti sembra più leggibile.

5.3 FUNZIONI VOID

Alcune funzioni non restituiscono alcun valore – in alcuni linguaggi di programmazione, funzioni di questo tipo prendono il nome di procedure o subroutines. Per permettere ciò, è necessario specificare come tipo di ritorno la parola chiave **void**. Ad esempio:

```
#include <iostream>
using namespace std;

void saluta()
{
cout << "Ciao, mondo!" << endl;
}
int main()
{
saluta();
return 0;
}
```

In caso di funzione che non restituisce un valore, non è necessario prevedere un istruzione di **return,** ma è possibile farlo per interrompere l'esecuzione della funzione in un punto qualsiasi.

5.4 PASSAGGIO DEGLI ARGOMENTI

5.4.1 PASSAGGIO PER VALORE E PER RIFERIMENTO

Considera questo codice:

```
#include <iostream>

void raddoppiaNumero(int numero)

{
    numero *= 2;
}
int main()

{
  int i = 1;
  raddoppiaNumero(i);
  std::cout << "Il valore di i e': " << i;
  return 0;
}
```


Il risultato dell'esecuzione è:

```
Il valore di i e' 1
```

Perché mai il valore di i non è raddoppiato? La risposta è semplice: il passaggio degli argomenti in C++ avviene per copia. Al momen-

to della chiamata della funzione, infatti, la variabile i viene copiata in un'altra variabile distinta, cosicché ogni cambiamento applicato a "numero" non si rifletterà in "i". Ma noi **vogliamo** cambiare il valore di i all'interno della funzione. Come fare? Il metodo più indicato è il passaggio per riferimento, ottenuto mediante l'indicazione di una reference come parametro. Si dice in gergo che i è stato passato "per riferimento". Il nostro esempio cambia così:

```
#include <iostream>

void raddoppiaNumero(int& numero)
{
    numero *= 2;
}

int main()
{
  int i = 1;
  raddoppiaNumero(i);

  std::cout << "Il valore di i e': " << i;
  return 0;
}
```

Il risultato dell'esecuzione, stavolta, è:

```
Il valore di i e' 2
```

5.4.2 RIFERIMENTI DI SOLA LETTURA

Passare i parametri per riferimento spesso è preferibile, perché non viene effettuata alcuna copia dell'oggetto, e un'operazione simile per classi complesse può essere molto pesante in termini di memoria e velocità di esecuzione. In alcuni casi, si vuole approfittare di

questa velocità, ma si desidera che la variabile **non possa cambiare valore** nel corso della funzione. Questo si ottiene dichiarando il riferimento come argomento costante:

```
void raddoppiaNumero(const int& numero)
{
    numero *= 2; // errore: numero è di sola lettura!
}
```

La ragione per cui un programmatore dovrebbe autolimitarsi può non apparirti molto chiara (basterebbe imporsi di non cambiare il valore di numero durante l'esecuzione), ma a volte i buoni propositi non vengono mantenuti per semplice distrazione, o perché non saremo noi ad implementare la funzione - ad esempio nel caso in cui questa sia virtuale (come vedremo nel capitolo 7).

5.4.3 ARGOMENTI DI DEFAULT

È possibile indicare un valore predefinito per gli argomenti, per esempio:

```
#include <iostream>

void saluta(char* messaggio = "Ciao, Mondo!")
{
    std::cout << messaggio << std::endl;
}

int main()
{
    saluta();
    saluta("Ciao, Universo!");
    return 0;
}
```

L'output dell'esecuzione sarà:

```
Ciao, Mondo!
Ciao, Universo!
```

In quest'esempio, nella prima istruzione di main non abbiamo specificato alcun argomento, pertanto il C++ ha usato il valore di default.

È possibile stabilire un numero arbitrario di parametri di default, purché vengano presentati **alla fine** (la ragione è facilmente comprensibile).

```
void eleva(int base, int esponente = 2); //giusto
void eleva(int esponente = 2, int base); //sbagliato!
```

5.4.4 SOVRACCARICAMENTO

Il sovraccaricamento (o **overloading**) delle funzioni permette di creare più funzioni con argomenti e comportamenti diversi, ma con lo stesso nome. Ad esempio:

```
void saluta()
{
cout << "Ciao, Mondo!";
}
void saluta(char* messaggio)
{
    cout << messaggio;
}
```

Confrontando quest'esempio con quello presentato in 5.4.2, si evince che gli argomenti di default sono un caso particolare di overloading.

Il sovraccaricamento delle funzioni è una particolarità del C++: in C sarebbe stato possibile creare qualcosa di simile solo scrivendo:

```
void saluta();
void salutaConMessaggio(char* messaggio);
```

Il che avrebbe richiesto al programmatore di porre costante attenzione a quando usare una funzione, e quando un'altra. Va tenuto conto che il sovraccaricamento delle funzioni introduce un overhead che può pesare sulle prestazioni, e anche la possibilità di ambiguità che il compilatore cercherà di risolvere secondo regole sofisticate - in caso di **eccessiva** ambiguità, il compilatore chiederà di specificare una funzione tramite casting esplicito. Se ben utilizzato, però, il sovraccaricamento è uno strumento senza pari, soprattutto in alcuni ambiti, come l'overloading degli operatori. Pensa a quanto sarebbe scomodo il **cout** se dovessi ogni volta prevedere una funzione diversa per scrivere un int, o un char, etc…

5.5 VARIABILI GLOBALI

Poiché la visibilità di una variabile si esaurisce all'interno della funzione, ne consegue che essa non sarà riconosciuta da altre funzioni. Ad esempio, il codice seguente è un errore:

```
#include <iostream>
void stampa()
{
std::cout << i; //errore: i non è visibile!
}

int main()
{
```

```
int i;
for (i=0; i < 10; i++)
        stampa();

return 0;
}
```

Per questo motivo, le variabili dichiarate all'interno di una funzione prendono il nome di **variabili locali.** Per simmetria, esiste la possibilità di definire della **variabili globali**, che siano visibili all'interno di tutte le funzioni.

```
#include <iostream>

int i;

void stampa()
{
std::cout << i; //giusto: i è globale
}

int main()
{
for (i=0; i < 10; i++)
        stampa();

return 0;
}
```

Questo codice funzionerà, dal momento che la **i** è stata dichiarata nello spazio globale, fuori (e prima) delle funzioni. Le variabili globali sembrano generalmente una grande invenzione ai neofiti e un insulto alla programmazione agli esperti. La ragione per cui le

variabili globali vanno evitate è semplice: nessuno può sapere quali funzioni accederanno e cambieranno la variabile, pertanto è impossibile essere sicuri che il suo valore rimarrà stabile. Una variabile locale, per contro, è soggetta soltanto al dominio della funzione, pertanto ha un comportamento perfettamente predicibile. Problemi di questo genere possono essere risolti, semplicemente prevedendo il passaggio di un argomento:

```
#include <iostream>

void stampa(int i)
{
std::cout << i;
}

int main()
{
for (int i=0; i < 10; i++)
        stampa(i);

return 0;
}
```


Come al solito, esistono rarissime eccezioni alla buona regola di evitare l'uso di variabili globali. Inoltre, è anche possibile dichiarare strutture, unioni e typedef (nonché classi) globali, per i quali l'operazione è invece perfettamente sensata.

5.6 PROTOTIPI

Avrai notato che finora ho sempre posto la funzione main per ultima. Non è un caso: la ragione è che la funzione main ha un vincolo di dipendenza dalla funzione stampa, perché la richiama al

suo interno. Se proviamo ad invertire l'ordine, come in:

```
int main()
{
a();
return 0;
}

void a() {}
```

il compilatore si lamenterà dicendo che **a** non è mai stata dichiarata. Infatti al momento dell'errore **non è ancora arrivato a leggerla!** Questa faccenda delle dipendenze può diventare molto seccante da analizzare, e nei progetti di dimensioni notevoli elaborare un ordine da seguire risulta impossibile, perché si verificano facilmente casi di ricorsione indiretta, come in:

```
void a() {b();}
void b() {a();}

int main()
{
a();
return 0;
}
```

In quest'esempio, comunque si rigirino queste funzioni non si riuscirà a trovare una configurazione possibile. Per questo motivo il C++ permette di dichiarare che **esisterà** una data funzione, che in seguito verrà definita in modo più compiuto.
Questo tipo di affermazione prende il nome di **prototipo** (o **dichiarazione**), e viene effettuata anticipando l'intestazione della funzione, senza specificare il codice relativo.

```
#include <iostream>

void stampa(int);

int main()
{
for (int i=0; i < 10; i++)
        stampa(i);

return 0;
}

void stampa(int i)
{
std::cout << i;
}
```

Come mostra l'esempio, nel prototipo non è necessario inserire il nome degli argomenti: per il compilatore è sufficiente sapere che esiste una funzione stampa, che prende un intero come argomento, non restituisce valore, e sarà definita più tardi. Così facendo, può interpretare correttamente il riferimento a **stampa(i)** in main.

5.7 FILES HEADER

Immagina un progetto medio-grande sulle 200.000 righe di codice: centinaia di funzioni, decine di strutture e tipi definiti, più qualche variabile globale. Anche col sistema dei prototipi, il risultato sarebbe un monolite ingestibile. I files dei progetti reali, invece, non superano quasi mai le duecento righe di codice, e molto più spesso si limitano a poche decine. . Il trucco sta nel **suddividere** i compiti in files appositi, utilizzando il meccanismo dei prototipi per separare

nettamente le dichiarazioni di funzioni, strutture e classi dalle loro definizioni. Le **dichiarazioni** vengono scritte in files chiamati **headers**, ai quali viene associata solitamente l'estensione .h, .hpp, o nessuna (ad esempio <iostream>). Le **definizioni,** invece, vengono poste nei file **sorgente (**con estensione .cpp), possono così far riferimento agli headers che servono loro, mediante le direttive #include. Nota che questa direttiva va seguita dal nome del file scritto:

- fra parentesi angolari, se questo va ricercato nella directory delle librerie (ad esempio <iostream>).

- fra virgolette, se invece va ricercato nella directory locale.

È esattamente il meccanismo che abbiamo utilizzato finora per utilizzare i flussi cin e cout della libreria <iostream>. L'esempio che segue mostra un tipico file header.

```
#ifndef SCRITTURA_H
#define SCRITTURA_H

const int MAX_COLONNE = 80; //numero massimo di colonne della
                                                        console

struct Scritta {
    char* testo;
    int   lunghezza;
};

Scritta creaScritta(char*, int); //inizializza una nuova scritta
void    scriviCentrato(Scritta); //scrive una scritta centrata

#endif
```

Si tratta di un file header per migliorare il sistema di stampa nella

console, che definisce un nuovo tipo di dati (scritta), che memorizzi una stringa stile C e la sua lunghezza. In una situazione normale avrei usato la comodissima **string** messa a disposizione dalle librerie standard, ma la definizione di un nuovo tipo rende quest'esempio più comprensibile. Il file header prevede la definizione di una **costante,** di una **struttura**, e di due **prototipi.** L'uso delle direttive al preprocessore #ifndef e #define è chiamato in gergo **sentinella**, e permette al file header di essere letto dal compilatore una volta sola: in seguito alla prima lettura, infatti, il preprocessore avrà già definito la macro SCRITTURA_H, e non analizzerà il contenuto del file.

5.8 NAMESPACES

La creazione di variabili esterne, costanti, funzioni, strutture, diventa un problema nei grossi progetti, per via del progressivo restringimento dello **spazio dei nomi**.

Dopo un centinaio di denominazioni, infatti, anche la fantasia più fervida si esaurisce, e bisogna far ricorso a denominazioni sterili e incomprensibili come quelle che affliggono molti progetti di stile c (esempio: cosa vorrà mai dire **wcsrtombs**?). Il C++ permette di risolvere questo problema utilizzando i **namespaces,** ovvero dei reparti separati nei quali è possibile immettere tutte le definizioni correlate ad un problema. Progetti medio-grandi dovrebbero sempre far ricorso a questi strumenti, così come il codice scritto per essere referenziato come libreria esterna. La definizione di un namespace è semplice:

```
namespace nome {
    dichiarazioni
}
```

ad esempio:

```
namespace matematica {
    //definizione di struttura
struct Frazione {
        int numeratore;
        int denominatore;
}

//definizione di funzione
Frazione riduciFrazione(Frazione) {/*...*/};
}
```

I namespaces sono costrutti aperti, cosicché è sempre possibile aggiungere dei nuovi elementi in altri files ad un namespace già esistente:

```
namespace matematica {
//aggiunge una nuova funzione al namespace
Frazione sommaFrazioni(Frazione a, Frazione b) {/*...*/};
}
```

Una volta creati i namespace con le opportune dichiarazioni e definizioni è possibile richiamare i membri che ne fanno parte, usando l'operatore ::, ad esempio:

```
#include "matematica.h"
int main()
{
    matematica::Frazione f;
    f.numeratore  = 10;
    f.denominatore    = 5;
    f = matematica::riduciFrazione(f);
    return 0;
}
```

Come già sappiamo, è possibile avvalersi della direttiva **using namespace**, per dare per scontato che vogliamo riferirci ad un particolare namespace. L'esempio precedente può quindi essere riscritto come:

```
#include "matematica.h"

using namespace matematica;

int main()
{
    Frazione f;
    f.numeratore  = 10;
    f.denominatore    = 5;
    f = riduciFrazione(f);
    return 0;
}
```

Questo meccanismo permette ai programmatori C++ di avere una grande libertà nella scelta dei nomi, mantenendo la comodità nella scrittura ed evitando ambiguità.

5.9 PRATICA: CREAZIONE DI UN NAMESPACE

Per collaudare tutte le informazioni che abbiamo appreso finora, creeremo un namespace molto semplice per scrivere in maniera più appariscente sulla console, secondo la traccia già fornita nel paragrafo 5.5.

5.9.1 CREAZIONE DEL FILE SCRITTURA.H

Il file scrittura.h conterrà tutte le definizioni necessarie, incluse nel namespaces Scrittura.

```
//file scrittura.h
#ifndef SCRITTURA_H
#define SCRITTURA_H

namespace Scrittura {
const int MAX_COLONNE = 80; //numero massimo di colonne

struct Scritta {
      char* testo;
     int   lunghezza;
};

Scritta creaScritta(char*, int); //inizializza una nuova scritta
void    scriviCentrato(Scritta); //scrive una scritta centrata
};

#endif
```

5.9.2 CREAZIONE DEL FILE SCRITTURA.CPP

Una volta scritto il file header, possiamo provvedere alla definizione delle due funzioni dichiarate, inserendole nel namespace Scrittura. Creiamo il file scrittura.cpp e digitiamo il seguente codice:

```
//file scrittura.cpp
#include "Scrittura.h"
#include <iostream>

namespace Scrittura {
Scritta creaScritta(char* testo, int lunghezza)
{
    Scritta risultato;
```

```
    risultato.testo      = testo;
    risultato.lunghezza  = lunghezza;

    return risultato;
}

void scriviCentrato(Scritta scritta)
{
    //aggiunge gli spazi necessari per la centratura
    for (int i=0; i<(MAX_COLONNE - scritta.lunghezza - 1)/2; i++)
        std::cout << ' ';

    std::cout << scritta.testo;
     }
};
```

5.9.3 CREAZIONE DEL FILE MAIN.CPP

Infine, è giunto il momento di collaudare il nostro namespace. Creiamo il file main.cpp e digitiamo il seguente codice d'esempio:

```
#include "Scrittura.h"

using namespace Scrittura;

int main()
{
    scriviCentrato(creaScritta("Ciao, mondo!", 12));
    scriviCentrato(creaScritta("Stiamo scrivendo centrato!", 26));

    return 0;
}
```

L'output dell'esecuzione dell'esempio è visibile in (**figura 5.1**).

5.10 ESERCIZI

Estendi il file scrittura.h, aggiungendo la routine incorniciaScritta, che circonda il testo di asterischi. Ad esempio.

```
**********
** Ciao Mondo! **
**********
```

Modifica la funzione creaScritta, in modo che riceva in ingresso solo la stringa e calcoli da sé la lunghezza. Suggerimento: ricordati che la fine di una stringa in stile C è contrassegnata dal carattere nullo '\0'

PARADIGMA A OGGETTI

Nel capitolo precedente abbiamo visto lo stile di programmazione procedurale, tipico del C, che il C++ riprende con qualche miglioria. In questo capitolo e nel successivo analizzeremo il **paradigma orientato agli oggetti (OOP),** e come il C++ lo implementi attraverso l'uso delle **classi.** Contrariamente alla "credenza popolare", la programmazione a oggetti non è affatto nuova nel mondo informatico, né è stata introdotta dal C++: basti pensare a l linguaggio Simula, che nel 1967 (informaticamente parlando, **ere geologiche** fa) proponeva un modello compiuto di OOP. Il C++ trae quindi i suoi fondamenti teorici dai modelli applicati in Simula prima, e in Smalltalk poi. La rappresentazione delle classi e delle loro relazioni è anch'essa cambiata nel corso del tempo, fino a stabilizzarsi (**forse**) nella notazione che useremo qui: l'UML (Unified Modelling Language [7]).

6.1 DICHIARAZIONE DI CLASSI

6.1.1 DICHIARAZIONE DELLA CLASSE

Il principio dal quale muove l'OOP è che ogni ente è rappresentabile mediante un oggetto, che a sua volta è la composizione o la derivazione di altri. La definizione formale degli attributi e del comportamento dell'oggetto è realizzata mediante una **classe**.
La parola chiave **class** definisce una classe ed ha la seguente sintassi:

```
class nome
{
    dichiarazione classe
};
```

Laddove **dichiarazione classe** comprende una serie di specifiche, dichiarazioni e definizioni di vari elementi.

6.1.2 DICHIARAZIONE DEGLI ATTRIBUTI

La dichiarazione più semplice di una classe, consiste nello specificare una serie di **attributi** che fanno parte dell'oggetto descritto:

```
class Frazione
{
    int numeratore;
    int denominatore;
};
```

6.1.3 DICHIARAZIONE DEI METODI

L'esempio che abbiamo visto non è nuovo: nello stile di programmazione procedurale, è possibile ottenere lo stesso risultato con una struttura.

```
struct Frazione
{
    int numeratore;
    int denominatore;
};
```

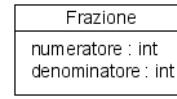

Figura 6.1: La struttura frazione.

cosicché in seguito possano essere definite delle funzioni che operano su una Frazione. Ad esempio:

```
void  stampa(Frazione frazione)
void   semplifica(Frazione& frazione);
double calcolaValore(Frazione frazione);
```

```
bool verificaValidità(Frazione frazione);
```

In questo caso, però, non c'è nulla a dimostrare sussista una relazione fra la struttura e le operazioni elencate, se non il passaggio dell'argomento di tipo Frazione. Mediante una classe, invece, è possibile definire anche delle funzioni (chiamate **metodi**) che agiscano sugli attributi:

```
class Frazione
{
    int numeratore;
    int denominatore;
    void   stampa();
    void        semplifica();
    double  calcolaValore();
    bool   verificaValidità() ;
};
```

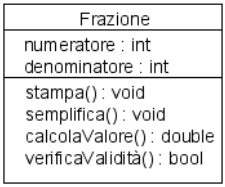

Figura 6.2: La classe frazione.

Questo rinforza notevolmente il vincolo fra una Frazione e le sue operazioni, e permette anche di eliminare il ridondante passaggio dell'argomento. Una volta definite correttamente le funzioni dichiarate, è possibile richiamarle semplicemente per mezzo dell'operatore punto (.), ad esempio:

```
int main() {
    Frazione f;
```

```
    f.numeratore   = 10;
    f.denominatore = 4;

    f.semplifica();
    f.stampa(); //mostra a video 5/2

    return 0;
}
```

6.1.4 LIVELLI DI VISIBILITÀ

Una classe può essere vista come una "scatola nera": una volta che il programmatore sa quali funzioni deve richiamare, non ha alcun interesse a conoscerne i meccanismi interni. E **non deve**: il principio secondo il quale al mondo esterno non dovrebbe mai essere lasciato un pericoloso accesso al funzionamento privato della classe si chiama **incapsulamento**, ed è uno dei fondamenti della programmazione a oggetti. Possiamo capire facilmente l'entità del problema tentando di costruire una classe Frazione che sollevi il programmatore dall'obbligo di dover semplificare il numeratore e il denominatore. Per realizzare ciò, la classe provvede automaticamente alla riduzione, ogni volta che il numeratore e il denominatore cambiano. Per avere notifica del cambiamento, è necessario fornire dei metodi (in genere indicati col prefisso **set**, cioè "**imposta**") per la ridefinizione dei membri, ad esempio:

```
class Frazione
{

    int numeratore;
    int denominatore;

    void setNumeratore(int numero)
```

```
{
    numeratore = numero;
    semplifica();
}

    void setDenominatore(int numero)
    {
        denominatore = numero;
        semplifica();
    }
};

Chi usa la classe può quindi gestirla in questo modo:

int main() {
    Frazione f;
    f.setNumeratore(10);
    f.setDenominatore(4);

//semplificazione automatica: f vale 5/2!

    f.stampa();
    return 0;
}
```


Ma chi ci garantisce che il programmatore non imposterà direttamente **numeratore** e **denominatore** senza usare le funzioni di set, vanificando tutti i nostri sforzi? La colpa, in un simile caso, non sarebbe sua, ma nostra: abbiamo progettato la classe violando il principio di incapsulamento, senza avvalerci dei **livelli di visibilità** che il C++ permette di definire per membri e funzioni di una classe. Possiamo dichiarare numeratore e denominatore come **membri privati**, in questo modo:

Figura 6.3: La classe frazione, con i

```
class Frazione
{
private:
    int numeratore;
    int denominatore;
public:
    void setNumeratore(int numero);
void setDenominatore(int numero);
int getNumeratore() {return numeratore;}
int getDenominatore() {return denominatore;}
};
```

In questa maniera, il programmatore che userà la classe Frazione non potrà più avere accesso diretto a numeratore e denominatore, che saranno adoperati solo internamente alla classe. Il C++ definisce tre livelli di visibilità possibili:

- **private:** indica che i membri/metodi che seguono non saranno accessibili all'esterno della classe;

- **protected:** indica che i membri/metodi che seguono non saranno accessibili all'esterno della classe, se non per le classi derivate;

- **public:** indica che chiunque avrà accesso ai membri/metodi che seguono.

Se non si pone alcun indicatore di accesso, il C++ darà per sottointeso che tutte le dichiarazioni sono **private.**

6.1.5 L'ATTRIBUTO *THIS

Talvolta può essere necessario dover specificare un puntatore all'istanza dell'oggetto su cui sta effettivamente operando la classe. Ciò è possibile attraverso la parola chiave **this**:

```
class Batterio
{
public:
    Batterio* padre;
    Batterio generaFiglio() {
        Batterio figlio;
        figlio.padre = this;
        return figlio;
    }
};
```

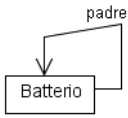

Figura 6.4: associazione riflessiva della

In questo caso, un Batterio è capace di generare un figlio che si ricorderà di chi è suo padre. Per ottenere ciò, al momento della creazione il padre trasmette il suo indirizzo al figlio mediante l'attributo **this.**

6.2 COSTRUTTORI

Torniamo alla nostra classe Frazione, che ha ancora un problema. Poniamo di scrivere questo codice:

```
int main() {

    Frazione f;
    //quanto valgono f.numeratore e f.denominatore?
    f.setNumeratore(10);

    return 0;
}
```

Alla riga f.setNumeratore(), il programma tenterà di semplificare la frazione, ma il denominatore non è ancora stato inizializzato! Ciò porta a risultati imprevisti. Per questo, deve essere fornito un metodo per l'inizializzazione dei dati, che venga richiamato in automatico alla dichiarazione della variabile. Questo metodo prende il nome di **costruttore.** Il costruttore è una funzione speciale che ha lo stesso nome della classe e non restituisce alcun valore.

6.2.1 COSTRUTTORE DI DEFAULT

L'esempio più semplice di costrutture è:

```
class Frazione
{
//...
public:
    Frazione() {};
//...
};
```

Qui non viene definita alcuna operazione per la funzione: questo è il **costruttore di default,** che viene assunto implicitamente quando non ne viene specificato uno personalizzato. Ciò, ovviamente, non risolve il problema della classe Frazione descritto nel paragrafo precedente.

6.2.2 COSTRUTTORE CON PARAMETRI

È possibile stabilire che il costruttore abbia un numero arbitrario di argomenti, ad esempio:

```
class Frazione
{
private:
    int numeratore;
    int denominatore;
public:
    Frazione(int n, int d)
{
numeratore = n;
denominatore = d;
semplifica();
};
};
```

In questo modo la dichiarazione di una Frazione deve sempre essere seguita dai due parametri, il che risolve il nostro problema di inizializzazione: una Frazione avrà sempre un numeratore e un denominatore ben definito.

```
int main() {
    Frazione f(10,5);
    f.stampa() // 2!
    return 0;
}
```

6.2.3 COSTRUTTORE CON PARAMETRI PREDEFINITI

Il costruttore è una di quelle funzioni che più spesso si avvantaggiano dell'uso di argomenti di default (vedi paragrafo 5.4.3).

Attraverso di essi, infatti, è possibile stabilire una serie di costruttori a vari liveli di specializzazione, con una sola scrittura. Ad esempio:

```
class Frazione
{
private:
    int numeratore;
    int denominatore;
public:
    Frazione(int n=1, int d=1)
{
numeratore = n;
denominatore = d;
semplifica();
};
};
int main() {
    Frazione f1;   //f1 = 1/1
    Frazione f2(2);      //f2 = 2/1
    Frazione f3(2,3);   //f3 = 2/3
    return 0;
}
```

In modo analogo, è anche possibile fare uso di costruttori sovraccaricati (vedi paragrafo 5.4.4).

6.2.4 COSTRUTTORE CON INIZIALIZZATORI

Poiché il lavoro del costruttore è spesso quello di ricevere in ingresso dei parametri ed associarli agli attributi della sua classe, il C++ fornisce uno strumento per svolgere più rapidamente tale compito: gli inizializzatori.
La loro sintassi è:

```
NomeFunzione(argomenti) : attributo1(espressione),
                                    attributo2(espressione) …
```

Applicando gli inizializzatori nella nostra classe, abbiamo.

```
class Frazione
{
private:
    int numeratore;
    int denominatore;
public:
    Frazione(int n=1, int d=1) : numeratore(n), denominatore(d)
{
semplifica();
};
};
```

Quando è possibile scegliere, è preferibile usare gli inizializzatori perché rendono più leggibile il codice e più rapida l'esecuzione del programma.

6.3 PRATICA: LA CLASSE VETTORE

Per fare un po' di pratica con l'uso delle classi, possiamo provare a definire una classe Vettore che si occupi di gestire in maniera più sicura le operazioni che avvengono su vettori di interi: inizializzando a zero ogni elemento, e impedendo l'accesso a membri non esistenti. Grazie all'uso della memoria dinamica, inoltre, si libererà il programmatore dal vincolo di stabilire dei valori costanti.

```
#include <iostream>

class Vettore {
```

```
private:
    int* elementi;
    int grandezza;
public:
    Vettore(int g) : grandezza(g)
{
        //crea i nuovi elementi
elementi = new int[grandezza];

        //inizializza ogni elemento a zero
        for (int i=0; i<grandezza; i++)
                                                    elementi[i] = 0;
    }
    int getGrandezza() {return grandezza;}
    int& Elemento(int pos)
    {
      if (pos < grandezza && pos >= 0)
      return elementi[pos];
      else {
        std::cout << "Errore: indice fuori dai margini";
        return elementi[0];
      }
    }
};
```

Puoi provare ad usare questa classe in progetti di test, come:

```
int main()
{
    Vettore v(10); //10 elementi
    v.Elemento(5) = 10;
    if (v.Elemento(5) == 10)
        std::cout << "Funziona!";
```

```
    return 0;
}
```

6.4 DISTRUTTORI

Il **distruttore** è una funzione speciale che si occupa di eliminare dalla memoria i vari attributi della classe in cui è definito. La dichiarazione di un costruttore segue le stesse regole di quella di un costruttore, ma viene preposto al nome il simbolo di complemento (~).

6.4.1 DISTRUTTORE DI DEFAULT

Il distruttore non ha mai argomenti, pertanto la forma comune per dichiararlo in una classe è:

```
class Classe {
    ~Classe() {};
}
```

Questo costruttore non fa assolutamente nulla, ed è dato per implicito se non ne si definisce uno.

6.4.2 COME FUNZIONA UN DISTRUTTORE

Solitamente nessuno invoca mai il distruttore di una classe direttamente. È il programma a richiamarlo, quando un oggetto esce dal campo di visibilità. Ad esempio:

```
#include <iostream>
class Querula {
    ~Querula()
{
std::cout << "Ahime! Sono stata distrutta!";
    }
```

```
}

int main()
{
    {
Querula q;
    }
    //q viene distrutta
return 0;
}
```

Alla fine del blocco, q uscirà dal suo campo di visibilità, e quindi sarà richiamato il distruttore, lanciando il relativo lamento.

6.4.3 QUANDO USARE UN DISTRUTTORE

Normalmente non c'è alcun bisogno di prevedere un distruttore, se non si usa memoria dinamica. In questo caso, invece, il distruttore serve a deallocare tutte le risorse che l'oggetto sta detenendo, per evitare memory leaks. Questo è proprio il caso della classe Vettore che abbiamo realizzato nel paragrafo 6.3: gli elementi vengono allocati nel costruttore, ma non vengono mai distrutti! Ciò porterà, alla lunga, all'esaurimento delle risorse disponibili. Per ovviare si potrebbe prevedere una funzione **svuota()** definita così:

```
class Vettore
{
    //...

    void svuota()
    {
delete[] elementi;
}
}
```

Ma ciò darebbe all'utente della nostra classe l'onere di invocare **svuota()** prima che sia troppo tardi, rendendo così la nostra classe altamente insicura: i programmatori, infatti, non stanno mai molto attenti a certe cose. E cosa succederebbe se qualcuno svuotasse il vettore, e poi pretendesse di scriverci sopra? Usando un distruttore, invece, la cancellazione degli elementi sarà effettuata in automatico, senza intervento dell'utente della classe. Anche se ciò basterebbe ai nostri scopi, è comunque una buona pratica impostare gli elementi potenzialmente pericolosi ad un valore neutro e verificare la correttezza dei puntatori con dei controlli. Non si è mai abbastanza cauti, quando si ha a che fare con la distruzione di memoria dinamica:

```
class Vettore
{

    //...

    ~Vettore()
    {
if (elementi) { //se il puntatore è valido
delete[] elementi; //elimina
grandezza = 0; //neutralizza
elementi  = 0;  //neutralizza
        }
}
}
```


6.5 COSTRUTTORE PER COPIA

Quando si ha a che fare con la memoria dinamica, i distruttori sono una delle prime funzioni da considerare, assieme ai **costruttori per copia.** Se, infatti, possono verificarsi dei memory leaks o dei danni in seguito alla mancata (o errata) distruzione di un

oggetto, va ancora peggio per quelle operazioni in cui un oggetto viene copiato in un altro, come nel caso seguente:

```
int main() {
    Vettore v1(2);
    Vettore v2 = v1;

    v1.Elemento(0) = 10;
    v2.Elemento(0) = 20;

    std::cout << v1.Elemento(0);
return 0;
}
```

Ci aspetteremmo un bel 10, ma il programma non sarà d'accordo, restituendoci:

```
20
```

Perché? In istruzioni di tipo "v1=v2" , il C++ prevede la copia automatica membro a membro di v1 in v2: quindi la grandezza di v2 sarà impostata a 2, e il puntatore v1.elementi al puntatore v2.elementi. Quest'ultimo assegnamento è evidentemente un errore, perché v1 e v2 si troveranno a condividere lo stesso segmento di memoria, e ogni operazione svolta sul primo si rifletterà sul secondo, causando gli effetti dimostrati nell'esempio. E per di più la distruzione di uno dei due comporterebbe un dangling pointer nell'altro. In casi come questo bisogna modificare il comportamento del costruttore per copia, definendone uno:

```
class Vettore
{
    //...
```

```
    Vettore(const Vettore& b) : grandezza(b.grandezza)
    {
     //crea i nuovi elementi
     elementi = new int[grandezza];

     //copia ogni elemento
     for (int i=0; i<grandezza; i++)
         elementi[i] = b.elementi[i];
    }
    //...
}
```

Questo costruttore è un sovraccaricamento di quello originale (che va lasciato com'è, ovviamente), e ne ricalca la struttura: crea della nuova memoria per gli elementi e li inizializza (stavolta copiando i valori dagli elementi di b). Se facciamo girare il programma dopo aver ridefinito il costruttore per copia, otterremo il risultato esatto: 10!

6.6 OVERLOADING DEGLI OPERATORI

6.6.1 OVERLOADING DEGLI OPERATORI

Una delle possibilità più interessanti del C++ è quella di poter ridefinire il comportamento degli operatori aritmetici, logici, bit-a-bit, di incremento, e in più l'operatore virgola (,), freccia (->), le parentesi (tonde e quadre) e gli operatori di memoria dinamica: new, new[], delete, e delete[]. Di default, la maggior parte di questi operatori sarà disabilitata (praticamente tutti, esclusi quelli relativi alla memoria), mentre potrebbe essere utile e logico definirne un comportamento. Un esempio evidente è quello della classe Frazione, in cui l'uso degli operatori aritmetici è più che indicato. Ecco un esempio di ridefinizione dell'operatore di moltiplicazione:

```
class Frazione
{
    //...
    Frazione operator*(const Frazione& b)
    {
        return Frazione(numeratore * b.numeratore,
  denominatore * b.denominatore);
}
}
```

Questo permette di scrivere correttamente codice simile al seguente:

```
int main() {
    Frazione v1(4,3);
    Frazione v2(2,3);
    Frazione v3 = v1 * v2;
    v3.stampa();
    return 0;
}
```

Il risultato sarà:

```
8/9
```

L'overloading degli operatori si effettua, quindi, dichiarando la funzione operator seguita dall'operatore che si vuole ridefinire, il cui prototipo cambia a seconda dei casi (se l'operatore è unario o binario, quali argomenti vuole, etc…).

6.6.2 OPERATORI E CASTING

Potremmo pensare di definire correttamente un'operazione di moltiplicazione un intero e una frazione, e fra una frazione e un intero. Ma se proviamo il codice:

```
int main() {

    Frazione v1(4,3);
    Frazione v2 = v1 * 5;

    v3.stampa();

   return 0;
}
```

scopriamo che il compilatore la esegue. E che il risultato è valido:

```
20/3
```

Il compilatore sembra aver capito qual è il significato di un intero in ottica frazionaria! Magia? Certo che no: è tutto merito nostro, e di quando abbiamo avuto l'ottima idea di definire il costruttore con parametri di default. Il compilatore, infatti, ha provato ad effettuare un casting implicito in questo modo:

```
Frazione v2 = v1 * Frazione(5);
```

Così facendo, ha richiamato il costruttore Frazione(numeratore=5, denominatore=1), e il resto è venuto di conseguenza. La chiave per ottenere operazioni corrette su dati di tipo diverso è dunque questa: realizzare più costruttori sovraccaricati in maniera da dare un ampio ventaglio di casting e cercando di evitare quanto più possibile situazioni di ambiguità.

6.6.3 QUANDO NON SI DOVREBBE SOVRACCARICARE

Sovraccaricare gli operatori è utile e anche divertente (i programmatori si divertono in modo strano), ma non bisogna esagerare. La buona

regola che andrebbe rispettata è quella di sovraccaricare un operatore soltanto quando così facendo si introduce un'effettiva comodità per il programmatore che userà la classe. Il rischio, altrimenti, è quello di ottenere oggetti che presentano una sintassi complessa e operazioni poco intuitive. Ad esempio: nella classe Vettore si potrebbe essere tentati di creare un operatore +, ma quale comportamento dovrebbe assumere? Il + sommerebbe ogni elemento di a con ogni elemento di b? Oppure concatenerebbe gli elementi di **b** a quelli di **a**? Quando una risposta a simili quesiti non è intuitiva, è meglio lasciar perdere l'overloading e fornire delle funzioni dal comportamento più chiaro:

```
Frazione sommaMembroAMembro(const Frazione &b);
Frazione concatena(const Frazione &b);
```

6.6.4 QUANDO SI DOVREBBE SOVRACCARICARE

Ridefinire il comportamento di alcuni operatori è fondamentale. Solitamente quando si crea una classe si definiscono:

Il costruttore principale e le sue varianti
Il costruttore per copia
Il distruttore
L'operatore == (uguale a)
L'operatore < (minore di)
L'operatore = (assegnamento)

La ragione per cui si definiscono i due operatori == e <, è che sono fondamentali quando la classe viene definita nei contenitori standard, per alcune operazioni (ad esempio il **sorting**), oltre ad avere un'utilità intrinseca. Ad esempio, definendo in Frazione:

```
class Frazione
{
```

```
//...
bool operator==(const Frazione& b)
{
        return (numeratore == b.numeratore &&
  denominatore == b.denominatore);
}
//...
}
```

Diventa possibile testare l'uguaglianza di due Frazioni:

```
int main() {
   Frazione v1(10,5);
   Frazione v2(2);

   if (v1==v2)
       std::cout << "Sono uguali!";

   return 0;
}
```


6.6.5 L'OPERATORE DI ASSEGNAMENTO

Un discorso a parte merita l'**operatore di assegnamento**, che troppo spesso viene confuso dai neofiti con il costruttore per copia.

```
int main() {
   Vettore v1(10), v2(10);   //costruttori normali
   Vettore v3 = v1;          //costruttore per copia
   v3 = v2;                  //assegnamento

   return 0;
}
```

L'esempio qui sopra riportato mostra tre operazioni distinte: in particolare nota che **il costruttore per copia opera su dati non ancora inizializzati**, mentre quando viene richiamato l'operatore di assegnamento, v1 è già stato creato. Questa differenza è fondamentale nel caso di uso della memoria dinamica, dal momento che bisognerà provvedere prima alla distruzione degli oggetti che non servono più, e successivamente alla copia.

```
class Vettore
{
     //...
     Vettore operator=(Vettore(const Vettore& b) : grandezza(b.grandezza)
     {
      //distrugge i vecchi elementi
 grandezza = b.grandezza;
 if (elementi)
delete[] elementi;

 //copia i nuovi elementi

       elementi = new int[grandezza];
       for (int i=0; i<grandezza; i++)
            elementi[i] = b.elementi[i];
       return *this; //costruttore per copia
     }
}
```

6.7 MODIFICATORI DI MEMBRI E ATTRIBUTI

Può essere utile avere degli elementi che appartengono alla classe, ma non alla singola istanza dell'oggetto. Membri e funzioni di questo tipo si dicono statici.

Nota: peraltro, che l'operatore di assegnamento è un'espressione (vedi paragrafo 2.7), che deve restituire il valore ottenuto.

6.7.1 COSTANTI STATICHE

Una costante statica è una costante definita a livello di classe, attraverso le parole chiave static const.

```
class FileGrafico
{
public:
    static const char* estensione = ".jpg"; //costante statica
};
```

Una volta definita una costante statica, è possibile richiamarla attraverso la sintassi: Classe::costante.

```
int main()
{
    char* file = "Disegno" + FileGrafico::estensione;
    //...
};
```

6.7.2 MEMBRI STATICI

Analogamente alle costanti, è possibile definire dei membri statici. Questi possono essere visti come delle variabili globali accessibili attraverso la classe, per mezzo dell'operatore '::'.

```
#include <iostream>

class Batterio
{
public:
    static int totale;
```

```
    Batterio() {
        totale++;
    }
    ~Batterio() {
        totale—;
    }
};
int Batterio::totale = 0; //Definizione della variabile

int main()

{
  Batterio b1, b2;
  std::cout << "I batteri attivi sono: " << Batterio::totale << std::endl;

  Batterio b3;
  std::cout << "I batteri attivi sono: " << Batterio::totale; << std::endl;
  return 0;
}
```

In quest'esempio un membro statico viene utilizzato per tracciare il numero totale degli oggetti Batterio istanziati nel corso dell'esecuzione. L'output del programma sarà:

```
I batteri attivi sono: 2
I batteri attivi sono: 3
```

6.7.3 FUNZIONI STATICHE

Questa classe soffre di problemi di visibilità: chiunque potrebbe accedere al totale e cambiarlo. Per risolvere situazioni del genere il C++ permette anche di definire delle **funzioni statiche,** come dimostra il codice che segue:

```
#include <iostream>

class Batterio
{
private:
    static int totale;
public:
    static int stampa_totale() {//funzione statica
        std::cout << "I batteri attivi sono: " << totale << std::endl;
    }
    Batterio() {
        totale++;
    }
    ~Batterio() {
        totale—;
    }
};

int Batterio::totale = 0; //definizione di totale

int main()
{
  Batterio b1, b2;
  Batterio::stampa_totale();
  Batterio b3;
  Batterio::stampa_totale();
    return 0;
}
```

In questo modo, totale è nascosto e verrà inizializzato solo al momento della definizione. La funzione statica stampa_totale() permetterà, invece, di stampare a video il messaggio riguardo al numero di batteri attivi.

6.7.4 FUNZIONI CONST

Le funzioni di una classe possono essere distinte logicamente in due categorie: quelle che alterano il contenuto dei membri (ad esempio, le funzioni di set), e quelle che lasciano la classe invariata (ad esempio, quelle di get). Queste ultime si definiscono funzioni costanti. Per specificare (al compilatore, agli utenti della classe, e anche a se stessi) che la funzione di una classe è costante, è possibile utilizzare la parola chiave const alla fine dell'intestazione:

```
class Frazione
{
    void stampa() const
    {
        cout << numeratore << " / " << denominatore;
        numeratore = 5; //errore!
}
};
```

Nell'esempio qui riportato, il compilatore si rifiuterà di accettare la funzione stampa(), dal momento che l'assegnamento al membro **numeratore** viola il vincolo di costanza.

6.7.5 MEMBRI MUTABLE

A volte il vincolo "una funzione è costante se non altera **nessun** membro" e troppo rigido. Logicamente, una funzione è costante "se non altera alcun membro che interessi all'utente della classe". Immaginiamo una classe **mouse** nella quale, per fini statistici, vogliamo registrare il numero di click effettuati dall'utente.

```
class Mouse
{
private:
    Programma* programma;
```

```
    int numeroClicks;

public:
    //...
    void click()
    {
        programma->click(this);
numeroClicks++;
    }
};
```

In questo caso, la funzione click è logicamente costante per il mouse: l'utente non considera il numero di click come una variabile fondamentale. Variabili di questo tipo vengono dette **mutable.**

```
class Mouse
{
private:
    Programma* programma;
    mutable int numeroClicks;

public:
    //...

    void click() const
    {
        programma->click(this);
numeroClicks++;
    }
};
```

Poiché numeroClicks è definito come **mutable,** la funzione **click()** può legittimamente qualificarsi come const.

6.7.6 MEMBRI VOLATILE

In casi eccezionali e in progetti di una certa portata (interfacciamento ad harware a basso livello, gestione di thread multipli, etc…), può accadere che certe variabili **cambino valore** indipendentemente da quanto stabilito dal nostro programma. Questo potrebbe causare problemi, se l'applicazione non ne è a conoscenza. Un modo per mettere all'erta il compilatore su questa possibilità è dichiarare il membro come **volatile,** come nell'esempio:

```
class Driver
{
private:
    volatile long valorePorta;

    //...
};
```

6.7.7 FUNZIONI E CLASSI FRIEND

Anche le regole più ferree hanno le loro eccezioni, e talvolta è difficile riuscire a portare a termine un compito senza chiudere un occhio. Ad esempio, dichiarare che un certo membro è privato per **tutte** le classi, in **ogni** funzione, a volte può rivelarsi scomodo. Quando la relazione fra due classi e (o fra una classe e una funzione esterna) è molto forte, è possibile dichiarare un vincolo d' "**amicizia**" attraverso la parola chiave **friend.** Questa pratica eccezionale, nonostante non violi il principio OOP dell'**incapsulamento**, viene vista di cattivo occhio da molti progettisti, che ritengono che una pletora di funzioni amiche indichi uno scarso impianto architetturale. Nonostante ciò, in alcuni casi le funzioni **friend** sono molto utili: l'esempio più comune è quello di permettere il sovraccarico dell'operatore << in ostream, per l'operazione di inserimento su flusso (ad esempio cout), e dell'operatore >> in istream per quella di estrazione.

```
class Frazione
{
private:
    int numeratore;
    int denominatore;
public:
    friend ostream& operator<<(ostream& stream, const Frazione& f);
};
ostream& operator<<(ostream& stream, const Frazione& f) {
  stream << f.numeratore << '/' << f.denominatore;
  return stream;
}
```

L'esempio mostra l'uso del vincolo di friendship. Ma fa anche riflettere sul fatto che, se la classe è ben progettata, la dichiarazione di friendship può talvolta essere superflua (Frazione ha metodi get ad accesso pubblico che rendono disponibili i membri per ogni classe).

6.8 ESERCIZI

Definisci l'operatore **[]** per la classe Vettore, cosicché sia possibile scrivere istruzioni del tipo:

```
Vettore v;
v[2] = 4;
cout << v[2];
```

- Definisci la classe **Complesso** per la gestione dei numeri complessi. Parte immaginaria e parte reale devono essere double.
- Definisci i costruttori con argomenti predefiniti (r=0, i=0);
- Definisci le principali operazioni (aritmetiche, uguaglianza)

- Definisci la classe Scacchiera che:
  Contenga 8x8 classi Casella.
  La classe Casella deve saper definire correttamente un pezzo degli scacchi (se è presente, che colore ha, di che tipo è) e le operazioni correlate.
  Contenga dei metodi per l'accesso alle caselle
  Ridefinisca l'operatore << per l'inserimento su stream del suo contenuto.

RELAZIONI FRA GLI OGGETTI

Abbiamo imparato nel capitolo precedente a definire un oggetto e il suo comportamento attraverso le classi. L'idea fondamentale che muove il paradigma a oggetti è che programmare in questo modo sia più facile, perché il codice viene diviso in segmenti chiari, indipendenti, autoesplicativi. Farsi l'idea che l'approccio OOP renda la programmazione una passeggiata, però, è un errore. Oltre a saper definire una classe (attività di per sé piuttosto semplice), è necessario anche saper analizzare correttamente la rete di relazioni che legano i vari oggetti: si tratta di una tela invisibile e dai filamenti sottili, che rischia continuamente di sfaldarsi fra le mani del programmatore o di invischiarlo fra i suoi intrecci. Qui mostrerò gli strumenti che il C++ mette a disposizione del programmatore per stabilire i vincoli di relazione fra gli oggetti, ma **come** strutturare precisamente un progetto è qualcosa che si può imparare solo da altre basi teoriche [8, 9], e dalla propria esperienza.

7.1 GESTIRE PIÙ CLASSI

7.1.1 SUDDIVISIONE IN FILES

Un progetto C++ è normalmente composto da molte classi (centinaia), che cooperano fra loro nelle maniere più svariate. È perciò necessario disporre metodo per gestirle che sia tanto rigoroso quanto semplice. La logica direbbe di utilizzare un file per ogni classe. Questo proposito però diventa improponibile all'aumentare del codice di ogni metodo. Molto più semplice è separare nettamente la **dichiarazione** della classe e dei suoi elementi dalla loro definizione, in maniera analoga a quanto visto al paragrafo 5.7.
Nel file header risiederanno, dunque, soltanto le dichiarazioni di classi comprendenti dichiarazioni di membri, di eventuali membri e costanti statici, e i vari prototipi dei metodi.
Ad esempio:

```
//file Classe.h
#ifndef _CLASSE_
#define _CLASSE_

class Classe
{
private:
    int membro1, membro2;        //dichiarazione membri
    static char statico; //dichiarazione membro statico
public:
    Classe(int m1, int m2); //dichiarazione costruttore
    ~Classe(); //dichiarazione distruttore
    int Metodo1(int& param); //dichiarazione metodo
    //...
};

#endif
```

Nel file di codice, invece, risiederanno le definizioni delle funzioni e inizializzazioni dei membri statici della classe. Per definirli è necessario ripeterne il prototipo (o la dichiarazione) preponendo all'identificativo il nome della classe seguito dall'operatore "::". Ad esempio:

```
//file Classe.cpp
#include "Classe.h"

char Classe::statico = 'M'; // inizializzazione membro statico

Classe::Classe(int m1=1, int m2=2) : membro1(m1), membro2(m2)
{
    //codice del costruttore
};
```

```
~Classe::Classe()
{
    //codice del distruttore
};

int Classe::Metodo1(int& param)
{
    //codice del metodo
}

//...

#endif
```

come si vede da questo esempio, al fine evitare ambiguità è necessario che i prototipi delle funzioni dei due file siano perfettamente coincidenti. il C++ definisce comunque alcune semplici regole per non duplicare il codice inutilmente:
Gli inizializzatori di un costruttore devono essere posti soltanto alla sua definizione (quindi, nel file cpp).
Gli argomenti di default devono essere specificati soltanto una volta (o nella dichiarazione, o nella definizione).
Le variabili dichiarate come statiche devono essere successivamente definite da qualche parte.
Nella definizione di variabili statiche non è necessario ripetere la parola chiave static (vedi esempio in 6.7.2)

7.1.2 FUNZIONI INLINE

Quando la definizione di un metodo o di un costruttore avviene all'interno del blocco class, e non in un file di definizione esterno, tale funzione viene chiamata **inline**, e viene trattata dal compilatore in modo differente. Bisogna tener conto che il meccanismo di chiamata a funzione implica un (minimo) dispendio di risorse:

l'ambiente di lavoro deve porre sullo stack gli argomenti, provvedere alla chiamata, e ciò richiede tempo. Per funzioni che svolgono un lavoro molto semplice (poche righe di codice), tutto ciò può costituire uno spreco: sarebbe sufficiente incollare le righe che compongono la funzione (a mo' di macro da preprocessore) evitando, così, la chiamata. Il compilatore svolge automaticamente questo lavoro per le funzioni inline, duplicando il codice ed evitando le chiamate relative – o, quantomeno, ci prova: in alcuni casi (ricorsività diretta o indiretta, per esempio) è semplicemente impossibile. Il programmatore può quindi inserire definire all'interno del blocco class delle piccole funzioni che si vuole subiscano questo trattamento. Se si vuole definirle all'esterno, è possibile usare la parola chiave **inline**. Se sei un po' confuso sul quando definire delle funzioni inline, tieni presente che pretendere di dare consigli a un software sofisticato come un compilatore è ridicolo, se non si ha una buona conoscenza delle tematiche relative, nonché della struttura dello stesso. I compilatori più competenti sul fronte dell'ottimizzazione interprocedurale sono in grado di identificare da soli le funzioni inline e di decidere automaticamente come richiamarle, in barba ai nostri **suggerimenti**: talvolta non vale la pena di complicarsi troppo l'esistenza.

7.2 AGGREGAZIONE E ASSOCIAZIONE

7.2.1 AGGREGAZIONE O COMPOSIZIONE

L'aggregazione è uno dei metodi più semplici per stabilire un vincolo (di tipo **hasa**: "ha un") fra due classi A e B: consiste nel creare un membro di tipo B in una classe di tipo A. Esempio:

```
class Interruttore
{
```

```
    //definizione di Interruttore
};

class LettoreStereo
{
    Interruttore interruttore;
    //...
};
```

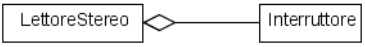

Figura 7.1: Relazione di aggregazione.

In questo esempio un **LettoreStereo possiede** un **Interruttore**: se il vincolo è molto forte (direi che è il caso) possiamo dire che il **LettoreStereo si compone** (anche) di un interruttore.

7.2.2 ASSOCIAZIONE

Spesso il vincolo di aggregazione è troppo stretto, e si vuole che due classi siano in relazione senza che una **possieda** letteralmente l'altra. In questo caso si può usare un vincolo di associazione, che si può stabilire in molti modi diversi.
Ad esempio usando un puntatore o un riferimento all'oggetto.

```
class Persona
{
    //definizione di persona
};
class LettoreStereo
{
    Persona* proprietario;
};
```

In questo caso, si è stabilito anche un vincolo di associazione fra un **LettoreStereo** e il suo **proprietario**. I vincoli di associazione godono di molta più libertà rispetto a quelli di aggregazione e composizione, pertanto vengono usati molto spesso nella programmazione C++.

7.3 EREDITARIETÀ

7.3.1 RELAZIONE DI TIPO ISA

Fra due classi è possibile stabilire un vincolo di tipo **isa (is-a: è un),** per mezzo dell'ereditarietà. Ad esempio, un programmatore, a suo modo, **è una** persona come le altre – se si esclude il fatto che conosce un linguaggio di programmazione e consuma caffè in quantità industriale. Pertanto è possibile definirlo così:

```
class Persona
{
    string nome;
    string cognome;
    //...
};

class Programmatore : public Persona
{
    string linguaggio;
    string caffèPreferito;
    //...
};
```

Il **Programmatore** eredita così tutti gli attributi di una persona, aggiungendo in più i membri e i metodi propri di un programmatore.

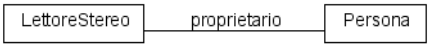

Figura 7.2: Programmatore deriva da Persona.

7.3.2 DICHIARAZIONE DI CLASSI DERIVATE

Per dichiarare che una classe **deriva** da un'altra, si usa la seguente estensione sintattica:

```
class ClasseDerivata : modificatoreAccesso ClasseBase
```

ModificatoreAccesso può essere **private**, **protected** o **public**, e stabilisce la nuova visibilità di cui godranno i membri e i metodi della classe base all'interno della classe derivata, in accordo con la tabella 7.1. Se non si specifica un modificatore di accesso, **private** sarà assunto per implicito**.**

Classe derivata	Classe base		
	private	protected	public
public	non accessibile	protected	public
protected	non accessibile	protected	protecded
private	non accessibile	private	private

Tabella 7.1: Comportamento dei modificatori di accesso delle classi ereditate.

Nota bene che in ogni caso i membri **private** della classe di base non sono disponibili nella classe derivata. Questo è necessario, per evitare che qualche programmatore malizioso possa essere tentato di ereditare da una nostra classe solo per poter accedere ai suoi membri privati, violando così il principio dell'incapsulamento.

7.3.3 OVERRIDING DEI METODI

Potremmo definire un semplice modello del comportamento di una Persona così:

```
using namespace std;

class Persona
{
private:
    string nome;
    string cognome;

public:
    void Saluta();
};

void Persona::Saluta() {
   std::cout << "Ciao! Mi chiamo " << nome << ' ' << cognome;
}
```

Un **Programmatore** non si limiterebbe a un saluto così generico, ma, dopo essersi presentato, ne approfitterebbe per tessere le lodi del suo Linguaggio Di Programmazione preferito e cercare di convertire la persona che ha di fronte alla Sua grazia. Per differenziare il comportamento, è sufficiente ridefinire (in gergo, sottoporre a **override**) il metodo **Saluta**, ridichiarandolo esplicitamente:

```
class Programmatore : public Persona
{
private:
    string linguaggio;

public:
    void Saluta();
};
```

Dopodiché, occorre ridefinirlo in maniera appropriata. E qui sorge

un problema: la classe **Programmatore** non può accedere ai membri nome e cognome, perché sono stati definiti in **Persona** come **privati.** Le soluzioni sono diverse:

- Ridichiarare nome e cognome come **protected**
- Creare delle funzioni di **get** di tipo pubblico
- Richiamare il metodo **Saluta** della classe **Persona**

Poiché sappiamo già cavarcela nei primi due casi, qui ti mostrerò il terzo:

```
void Programmatore::Saluta() {
Persona::Saluta();
std::cout << " e programmo in " << linguaggio
   << ". Non lo conosci? Ma se e' l'Unico Vero Linguaggio...";
}
```

Nell'esempio si è usato il nome della classe base seguito dall'operatore '::' per indicare che volevamo accedere a quel metodo, e non al Saluta del Programmatore (che avrebbe generato una chiamata ricorsiva). Questo è un comportamento standard da usare ogniqualvolta vogliamo riferirci a un membro o ad una funzione della classe di base. Una volta definite queste classi, possiamo provare un progetto di test:

```
int main()
{
    Persona        Pinco;
    Programmatore    Tizio;
    Pinco.Saluta(); cout << endl;
    Tizio.Saluta();  cout << endl;
    return 0;
}
```

```
Ciao! Mi chiamo
Ciao! Mi chiamo e programmo in . Non lo conosci? Ma se e' l'Unico Vero
Linguaggio...
```

7.3.4 EREDITARIETÀ E COSTRUTTORI

Il risultato dell'ultimo codice d'esempio è un po' incompleto. La colpa è del fatto che non abbiamo implementato dei costruttori adatti a definire una Persona e un Programmatore in maniera conveniente. Per quanto riguarda la Persona, questo non è un problema:

```
class Persona
{
    //...
    Persona(string n, string c) : nome(n), cognome(c) {};
};
```

Potremmo chiederci se questo costruttore viene ereditato nella classe derivata, al pari di tutti gli altri metodi. La risposta è **no**: nessun costruttore viene mai ereditato (così come l'operatore di assegnamento). Occorre, quindi, definirne uno nuovo. Proviamo così:

```
class Programmatore : public Persona
{
    //...
    Programmatore(string n, string c, string l) //errore!
: nome(n), cognome(c), linguaggio(l) {};
};
```

E scopriamo che non funziona. La ragione è semplice: nome e cognome sono membri privati nella classe base, e pertanto inaccessibili (vedi 7.3.2). L'unica via per riuscire ad inizializzarli è richiamare il costruttore della classe di base:

```
class Programmatore : public Persona
{
    //...
    Programmatore(string n, string c, string l) //ok!
: Persona(n,c), linguaggio(l) {};
};
```

Questo codice è corretto. Ed è anche l'unico possibile: l'alternativa di richiamare il costruttore all'interno del codice che definisce la funzione non è valida, perché a quel punto **la classe di base è già stata costruita.** Occorre dunque ricordare che **il costruttore di una classe di base si richiama sempre come parametro fra gli inizializzatori**.

7.3.4 CLASSI DERIVATE DI CLASSI DERIVATE

Il mondo è bello perché è vario. E il C++ è bello perché permette di rappresentarlo facilmente: grazie al meccanismo dell'ereditarietà possiamo definire ogni tipo di persona, e per ognuna prevedere un saluto di tipo differente. Ma possiamo anche definire una persona di tipo specializzato, ad esempio:

```
class ProgrammatoreC64 : public Programmatore
{
public:
    ProgrammatoreC64(string n, string c) //ok!
        : Programmatore(n, c, "Basic") {};
    void Saluta() {
cout << "10 PRINT \"";Persona::Saluta();
cout << "\"" << endl
    << "20 GOTO 10";
    };
};
```

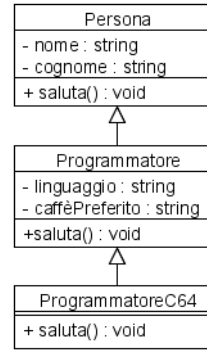

Figura 7.4: ProgrammatoreC64 deriva da Programmatore, che deriva da Persona

Il benvenuto sarà comprensibile solo da chi ha passato la giovinezza sulla schermata blu di un Commodore 64, ma quel che è evidente è che possiamo definire **classi derivate di classi derivate**, e andar sempre più specializzando a nostro piacimento, richiamando le definizioni delle classi base che ci fanno comodo (in questo caso, il **nonno** Persona::Saluta()). Possiamo provare a far girare un programma di prova:

```
int main()
{
    Persona         Pinco("Pinco", "Pallino");
    Programmatore   Tizio("Tizio", "Caio", "C++");
ProgrammatoreC64    Sam("Sam", "Pronyo");

    Pinco.Saluta(); cout << endl << endl;
    Tizio.Saluta(); cout << endl << endl;
    Sam.Saluta();   cout << endl << endl;

    return 0;
}
```

```
Ciao! Mi chiamo Pinco Pallino

Ciao! Mi chiamo Tizio Caio e programmo in C++. Non lo conosci?
Ma se e' l'Unico Vero Linguaggio...

10 PRINT "Ciao! Mi chiamo Sam Pronyo"
20 GOTO 10
```

7.3.5 SLICING

Analizziamo questo codice:

```
int main()
{

    Persona* tizio = new Programmatore("Tizio", "Caio", "C++");
    tizio->Saluta();
    delete tizio;

    return 0;
}
```


È interessante vedere come il compilatore interpreta la prima riga di **main**. A destra abbiamo creato un **Programmatore**, ma dall'altra parte lo stiamo puntando come **Persona**. A meno di cast, sappiamo che è illegale far riferimento ad oggetti di tipo differente da quello definito per il puntatore: ma un **Programmatore è una Persona**, in quanto classe derivata! Infatti il codice funziona. Solo che **tizio** ha perso gran parte del proprio smalto, ha smesso i panni del programmatore e si comporta in maniera piuttosto grigia e monotona. Ecco il suo saluto:

```
Ciao! Mi chiamo Tizio Caio
```

È un comportamento che ci aspetteremmo da una **Persona** normale. Questo è il senso dell'operazione della prima riga. Tizio punta all'interfaccia di tipo **Persona** del **Programmatore**. Il resto non è accessibile: né i membri, né i metodi, né i vari overrides. È un aspetto di un problema noto con il nome di **slicing**: una fetta dell'informazione e dei metodi della classe viene tagliata via. È un comportamento del tutto normale e spesso è quel che vogliamo, ma bisogna esserne coscienti, se si vogliono evitare brutte sorprese (attenzione ai costruttori per copia e agli operatori di assegnamento, per esempio!).

7.3.6 POLIMORFISMO

L'idea di puntare un oggetto di una classe derivata mediante un puntatore di tipo base ha i suoi vantaggi. Liste, contenitori, vettori e collezioni funzionano solo su oggetti dello stesso tipo. Questo ci permette di scrivere codice come:

```
int main()
{
Persona* persone[3] = {
 new Persona("Pinco", "Pallino"),
 new Programmatore("Tizio", "Caio", "C++"),
 new ProgrammatoreC64("Sam", "Pronyo")
};



    //Stampa i saluti
    for (int i=0; i<3; i++) {
        persone[i]->Saluta();
        cout << endl << endl;
    }

    //Distrugge le Persone
```

```
    for (int i=0; i<3; i++)
{
        delete persone[i];
    }

    return 0;
}
```

È evidente il vantaggio di questa scrittura: l'inizializzazione dell'array può essere effettuata durante il corso dell'esecuzione del programma (in gergo: in **late binding**): non è richiesto che sia già stato stabilito via codice in maniera statica di che tipo si vogliono le persone. Queste stanno comodamente in collezioni o liste, e possono essere oggetto di routine come il ciclo for. Il problema è che il comportamento degli elementi è sempre uguale:

```
Ciao! Mi chiamo Pinco Pallino
Ciao! Mi chiamo Tizio Caio
Ciao! Mi chiamo Sam Pronyo
```

Se riuscissimo a puntare i vari oggetti con un puntatore di un tipo di base, preservando al contempo i comportamenti definiti nelle classi derivate, raggiungeremmo uno degli obiettivi più utili e entusiasmanti dell'OOP: il **polimorfismo.**

7.3.7 FUNZIONI VIRTUALI

La via per far si che una classe abbia comportamento polimorfico è dichiararne un metodo come **funzione virtuale** per mezzo della parola chiave **virtual** (la quale si specifica solo nella dichiarazione). Un metodo virtuale è una funzione la cui invocazione viene risolta durante l'esecuzione, tramite un puntatore viene indirizzato sulla classe stabilita al momento della creazione dell'oggetto.

```
class Persona
{
    //...
    virtual void Saluta();
};

void Persona::Saluta() {
   std::cout << "Ciao! Mi chiamo " << nome << ' ' << cognome;
}

class Programmatore : public Persona
{
//...
public:
    virtual void Saluta();
};
//... eccetera
```

L'aggiunta di questa parola magica sarà sufficiente a produrre l'output:

```
Ciao! Mi chiamo Pinco Pallino
Ciao! Mi chiamo Tizio Caio e programmo in C++. Non lo conosci?
Ma se e' l'Unico Vero Linguaggio...
10 PRINT "Ciao! Mi chiamo Sam Pronyo"
20 GOTO 10
```

Abbiamo ottenuto un **comportamento polimorfico!**

7.3.8 FUNZIONI VIRTUALI NON DEFINITE

Nota che ridefinire una funzione dichiarata come virtuale è una possibilità, non un obbligo: nel caso in cui non la si definisca, viene utilizzata quella fornita dall'ultima classe base. Ad esempio:

```
class ProgrammatoreJava : public Programmatore
{
public:
      ProgrammatoreJava(string n, string c)
         : Programmatore(n, c, "Java") {};
};

 in questo caso, una richiesta simile:

Persona* joe = new ProgrammatoreJava("Joe", "Vattelappesk");
joe->Saluta();
```

sarà soddisfatta così:

```
Ciao! Mi chiamo Joe Vattelappesk e programmo in Java.
Non lo conosci? Ma se e' l'Unico Vero Linguaggio...
```


7.4 CLASSI ASTRATTE

A ben pensarci, nessuno è una **Persona** anonima: ogni essere umano ha degli interessi, un lavoro, delle caratteristiche che lo differenziano da un'entità amorfa. Se si accetta questa teoria, allora si dovrà considerare il termine **Persona** come un **concetto astratto**. Dal punto di vista della programmazione, l'equivalente di un concetto astratto è una classe che presenti almeno una **funzione virtuale pura**, ovverosia che dichiari una funzione come virtuale, ma rinunci a definirla – il che è diverso dal non darne una definizione, come in 7.3.8. Una funzione virtuale pura indica precisamente che quel metodo non deve esistere (e quindi neanche essere derivato da un'eventuale classe base), e lo rende noto tramite un assegnamento alla costante 0.

```
class Persona
```

```
{
    //...
    virtual void Saluta() = 0;
};
```

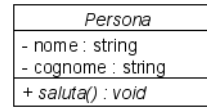

Figura 7.5: Persona è una classe astratta.

Classi di questo tipo si dicono **astratte**, e non possono essere istanziate. Per intenderci, in base alla definizione di **Persona** data qui sopra, questo codice è illegale:

```
int main()
{
    Persona Tizio("Tizio", "Caio"); //errore
}
```

Il compilatore si rifiuterà di proseguire, per il valido motivo che **Persona()** definisce **Saluta()** come una funzione virtuale pura, e pertanto è una classe astratta. Le classi astratte vengono quindi usate per fornire un modello che le classi derivate sono tenute ad adottare. Non è raro vedere classi completamente astratte, cioè formate unicamente da funzioni virtuali pure. In altri linguaggi, classi di quest'ultimo tipo prendono il nome di **interfacce**.

7.5 EREDITARIETÀ MULTIPLA

Il C++ permette l'**ereditarietà multipla**, ovverosia accetta che una classe possa derivare da due o più classi contemporaneamente.
Ad esempio:

```
class Macchina
{
    void Accendi();
    void Spegni();
    void Usa();
    //...
};
```

Se crediamo ancora al sogno dell'**IA forte**, potremmo definire un **Androide** come un essere che coniuga gli attributi e i comportamenti delle **Persone** con quelli delle **Macchine**.

```
class Androide : public Macchina, public Persona
{
    //...
};
```

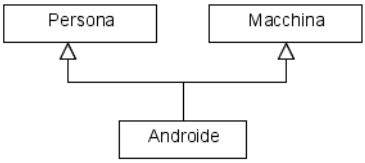

Figura 7.6: Ereditarietà multipla.

Alcuni linguaggi di programmazione non permettono questo genere di ereditarietà, per il fatto che potrebbero verificarsi dei fenomeni di ambiguità: se la Macchina e la Persona avessero entrambe un membro chiamato **nome**, quale dovrebbe ereditare la classe derivata? Come penso di aver ormai fin troppo sottolineato, il C++ non limita mai il programmatore solo perché qualcosa **potrebbe portare** a situazioni di inconsistenza, classi fragili o comportamenti pericolosi: sta a chi progetta l'applicazione creare delle gerarchie di classi basate prive di ambiguità, basate su modelli di grafi orientati aciclici.

7.6 ESERCIZI

Realizza la classe Punto2D capace di:

- Descrivere un punto nelle coordinate x,y
- Calcolare **distanza()** da un altro punto
- Supportare le operazioni aritmetiche e di incremento

Realizza la classe Punto3D, identica a Punto2D, ma con in più la coordinata z.

> Suggerimento (ovvio): usa l'ereditarietà!
> Suggerimento: un Punto3D può essere costruito a partire da un Punto2D, come (x,y,1)

INDICE

Tipi avanzati

Paradigma procedurale

Paradigma a oggetti

Relazioni fra gli oggetti

i libri di

ioPROGRAMMO

IMPARARE C++

Autore: Roberto Allegra

EDITORE
Edizioni Master S.p.A.
Sede di Milano: Via Ariberto, 24 - 20123 Milano
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

Realizzazione grafica:
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)

Resp. grafico: Paolo Cristiano
Coordinatore tecnico: Giancarlo Sicilia
Illustrazioni: Mario Veltri
Impaginazione elettronica: Lisa Orrico

Servizio Clienti
☎ **Tel. 02 831212 - Fax 02 83121206**
@ **e-mail: servizioabbonati@edmaster.it**

Stampa: Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Febbraio 2006